Optimizing Distributed Read-Only Transactions Using Multiversion Concurrency

Dan R. K. Ports Austin T. Clements Irene Y. Zhang
{drkp,amdragon,iyzhang}@mit.edu

December 15, 2007

Abstract

Distributed transactional systems typically achieve efficiency by abandoning true serializability for weaker forms of consistency that are difficult to reason about because they expose the concurrency in the underlying system. We explore an alternate route: weakening causality *instead of consistency. Our proposed algorithm achieves global serializability by sacrificing global causality, which we argue is reasonable in many situations. This allows our algorithm to achieve efficiency by permitting read-only transactions to operate on stale but locally available cache data. We present the details of a transactional block storage protocol that implements this form of concurrency control, as well as a performance evaluation of an experimental implementation of this protocol and comparison against conventional optimistic concurrency control.*

1 Overview

Many distributed systems, including distributed databases and transactional file systems, need to provide support for distributed transactions. Unfortunately, achieving full serializable isolation for distributed transactions is an expensive proposition, generally requiring locking, optimistic concurrency, and/or two-phase commit. The high cost of these protocols leads many implementors to consider alternatives that trade off consistency for performance.

Achieving full consistency is expensive because it encompasses two requirements, *isolation* and *freshness*. To ensure that the results of concurrent transactions are equivalent to a serial ordering, each transaction must operate on a transactionally consistent state; this requires either the use of two-phase locking or a multiversion scheme. Ensuring that transactions operate on fresh data limits the amount of cached information that can be used.

Existing work on weak consistency has focused on weakening isolation to levels less than full serializability [13, 7, 5]. Though these approaches are common in practice, allowing shorter-duration locks or fewer aborted transactions, it is difficult to reason about when they are correct [11]. Following a suggestion by Liskov and Rodrigues [18], we take an alternative approach, weakening *causality* instead of serializability.

Specifically, our approach allows read-only transactions to run on slightly stale data, but guarantees that they see a transactionally-consistent state. Read/write transactions are executed using a standard optimistic concurrency protocol, ensuring that they operate on the latest data. Allowing read-only transactions to use stale data not only makes it possible to avoid conflicts involving read-only transactions, but makes it possible to take advantage of data that is already in the cache. To ensure that the data seen by read-only transactions is reasonable, we require that it reflect the results of all transac-

tions that committed on the local node, and all other transactions that committed more than ϵ seconds ago; these constraints are specified formally in Section 3.1.

We argue this is a reasonable model for users of the system, since many transactions do not need to be run on the latest version of data as long as they still see a consistent state. As a result of our relaxed freshness requirements, the anomalies that can occur are ones where a read-only transaction A executed on one node fails to see the results of a transaction B that recently committed on a different node. However, although B might have happened before A, the commit point could just as well have happened slightly later, after A. Changing the ordering has no visible effect as long as there has been no communication between the two nodes involved to synchronize them; the two transactions can be said to happen concurrently and thus can be reordered.[1] We assume that the staleness interval ϵ is small enough that a synchronizing communication between nodes is unlikely to happen in the interval, but a more complete solution would use Lamport clocks [16] to explicitly model synchronization and concurrency in the system.

Our system optimizes the performance of read-only transactions. Such transactions are quite common in file systems, where the vast majority (over 80%) of operations are read-only [27, 20]; our experiments also confirmed this property. Moreover, it also eliminates potential conflicts between read-only and read/write transactions, and reduces concurrency overhead for read-only transactions. It is necessary for transactions to specify whether they are read-only or read/write when they begin, but this requirement is not onerous; a simple static analysis can usually determine whether it is read-only.

Besides weakened causality, the principal cost incurred by our scheme is additional storage space. However, this cost should be small, because only a few extra revisions will need to be kept. Moreover, storage capacity is rarely the limiting factor for scalability in such systems. Indeed, a number of systems, including temporal databases [25, 21], persistent object stores [19], and versioned file systems [24, 14] already maintain a history of versions in order to enable queries on previous system state, so such systems would incur no additional storage cost for our protocol.

The structure of this paper is as follows: Section 1 provides an overview of the system, ending with this paragraph giving an outline of the paper. Section 2 describes the architecture of the system and the interface provided to clients. Section 3 explains our concurrency control protocol in detail, explaining what properties it guarantees and how it achieves them. Section 4 describes our implementation, and Section 5 analyzes its performance using file system traces. We review related work in Section 6, and describe our plans for future work in Section 7. Finally, Section 8 concludes.

2 System Model

Our system exposes a transactional, distributed block store interface. The generality of a block store allows it to serve as the storage layer for a range of applications, from distributed database systems to transactional file systems, etc.

As shown in Figure 1, a centralized block server provides primary storage for block data. Each client application connects to the server through a standard client library. This library presents a procedural transactional block store interface to the application and manages communications with the server as well as client-side caching.

While frontend applications could be built directly atop the client library, the library interface is intended for backend applications, which then re-expose some application-specific interface to the local host. For example, a database engine could be built atop the library, which would then expose a SQL interface to local database clients.

[1] This is the same as the distinction between Lamport's *happened-before* and *happens-before* relations [16].

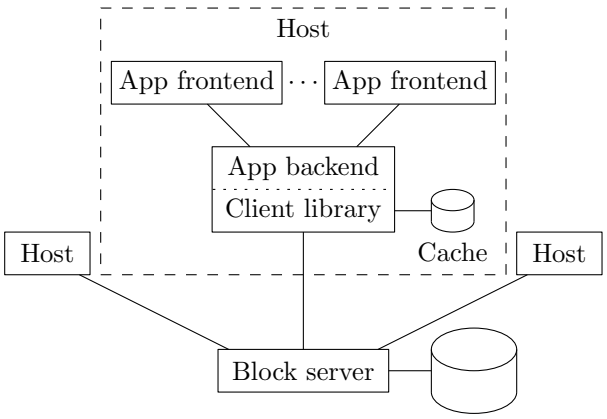

Figure 1: System model

Alternatively, a file system service could be built atop the library, which would then expose a file system interface to local applications through some kernel mechanism.

In addition to taking better advantage of per-host resources, this client-per-node structure may be necessitated by the causality requirements of the application. In particular, causality is guaranteed for all operations performed through a particular instance of the client library, but not between instances, even if running on the same host. For example, a file system that created an instance of the client library for each application would expose the user of the file system to violations of causality, even on a single host. Thus, the design of any application of the block store would have to account for the application's causality requirements. It may be possible to provide more flexible causality guarantees, but this is future work.

The client library itself presents a fairly straightforward block store interface. A block store object acts as a factory for read/write and read-only snapshots, which provide methods for reading blocks and committing and (in the case of read/write snapshots) writing blocks, creating new blocks, and aborting. Any operation on a read/write snapshot can potentially abort, so applications must be prepared to handle this possibility.

3 Algorithm

3.1 Properties

Specifying the desired consistency properties of our system requires introducing a few new definitions. The first requirement we set forth is a standard one:

Property 1 (Global Serializability)**.** *There exists a global serial ordering of all transactions such that the results of all transactions are consistent with this ordering.*

Note that, although a serial ordering exists, we have not stated *what* that serial ordering may be. In particular, the serial schedule may not correspond to the wall-clock-ordering of transaction commit times.

We require that the staleness of data seen by a transaction can be limited. (This also eliminates several vacuous solutions!) Specifically:

Property 2 (Freshness)**.** *A read-only transaction sees a consistent state of the database at a time no earlier than ϵ before the time it began, where ϵ is a property of the transaction.*

If a single user (or program) ran a succession of transactions, and their results were executed out of order, chaos would ensue, so we need the following property:

Property 3 (Local Causality)**.** *A transaction will see the effects of all transactions on the local node that committed before it started.*

This local causality property stands in contrast to the typical causality property of distributed transaction systems, where a transaction will see the effects of *all* transactions that committed before it started. Section 1 argues that this is a reasonable property.

Finally, our protocol allows the following nice property:

Property 4 (Conflict-freeness)**.** *Read-only transactions are never aborted, and can be executed without blocking for another transaction.*

Knowing that read-only transactions cannot be aborted can greatly simplify programming, since there is no need to be prepared to retry the transaction.

3.2 Protocol

The client-server protocol divides into three separate, but interacting sub-protocols: A protocol for filling and maintaining client-side caches, a protocol for committing read/write transactions, and a protocol for assigning read-only transaction timestamps.

3.2.1 Cache Coherence

The server maintains a multiversion block store, such as the one depicted in step (1) of Figure 2. Each block in the block store consists of a set of *versions*, each of which is valid over some non-overlapping range of time. The current version of a block is *unbounded*, meaning that its upper bound is unknown until a write to that block installs a new version at some later time.

Each client maintains a local cache with a structure similar to that of the server's block store. This cache provides two operations for reading versions:

- *get*(*blockid*, *timestamp*), which retrieves the contents of the block version whose range includes the given timestamp. This may be the current version or a historical version of the block.
- *getLatest*(*blockid*, *transaction*), which retrieves the current version of a block.

If the requested version is not present in the client-side cache, it forwards the request to the server, which responds with the contents and time range of the appropriate version. A typical historical request can be seen in step (2) of Figure 2. If the requested version turns out to be unbounded, then the server will add the client that issued the request to *holder set* for that block. In the example in Figure 2, this happens for block B in steps (3) and (4).

Because of the block store's multiversion nature, a given version is essentially immutable with the one proviso that an *unbounded* version can become a *bounded* version when its upper bound becomes fixed. The holder set of a block tracks exactly which clients have an unbounded version of a block in their cache so that when that version becomes bounded, the server can immediately notify these clients of this change via a *deprecation* message. For example, step (5) of Figure 2 shows the deprecation message sent to client 1 when a new version of B is installed at timestamp 4.

3.2.2 Read/Write Transactions

Read/write transactions use a protocol based on a mix of standard optimistic concurrency control [15] and multiversion concurrency control [8]. Block reads are satisfied optimistically from the latest, unbounded version of blocks in the client's cache whenever possible. Block writes go to a per-transaction side-store. Upon commit, the read and write sets of the transaction are sent to the server, which

1. Validates that all of the block versions read by the transaction are still unbounded. This ensures that the transaction can be placed next in the serial ordering because it is consistent with the latest version of the block store.
2. Assigns the transaction a timestamp α one greater than that of previously committed transaction. The timestamp indicates that the transaction read from the block store at time α and created the contents of the block store at time $\alpha + 1$.
3. Installs the blocks from the transaction's write set at time $\alpha + 1$. If this causes existing block versions to become bounded, then the server will send out deprecations, as described in the previous section.

This write protocol ensures global serializability of read/write transactions.

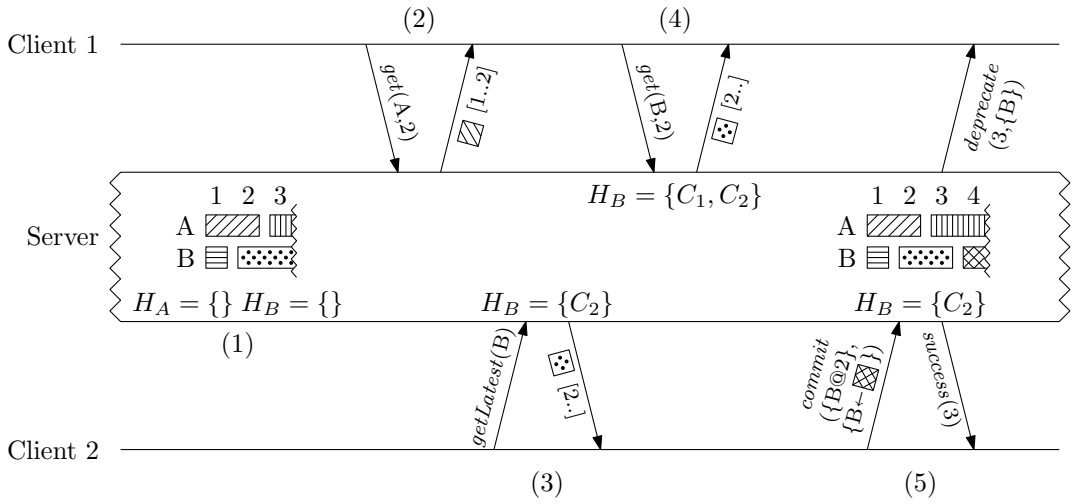

Figure 2: Protocol example with two clients. (1) Initially, the server is storing two blocks spanning times 1 to 3. Both blocks have a single historical version and no clients are in either block's holder sets. (2) Client 1 requests A at time 2. The server replies with the historical contents of A and the time range of that version. (3) Client 2 requests the latest version of B. The server replies with the current contents and time range of B. Client 2's cache will now contain an unbounded copy of B, so the server adds client 2 to the B's holder set. (4) Client 1 requests B at time 2. This is the current version of B, so the server adds client 1 to B's holder set. (5) Client 2 performs a commit, indicating which version of B it read and proposing a new version of B. After validating the commit request, the server replies with the time assigned to the transaction. Because the latest version of B was replaced, the server removes all of the clients from the holder set except the committing client and sends deprecations to these clients.

When creating new blocks, the client generates a *random* block ID, which it transmits to the server with the commit request. If the block ID collides with an existing block ID, then the server will abort the request. Otherwise, the block write proceeds as usual. Block ID's are considered sparse enough that this should rarely occur in practice. Our implementation, for example, uses 64-bit integers, making the probability of a conflict negligible.

Our algorithm goes beyond the standard validate-on-commit approach of optimistic concurrency control by employing *active aborts*. Because the cache is actively kept coherent by asynchronous deprecation messages, the validation condition given above can be checked continuously on the client side. Specifically, if a deprecation arrives for a block in an active transaction's read set, then it is impossible for that transaction to pass validation, so it can be aborted as soon as possible. Validation on commit is still necessary, however, as a critical deprecation message and a commit message may have crossed paths on the network; however, once a transaction requests a commit, it is very likely that the commit will succeed simply because the window during which such a conflict could arise is very small.

The standard optimistic approach of waiting until commit time to detect conflicts has a number of drawbacks. First, by waiting to abort a transaction, work is wasted in the event of a conflict [17]. Second, in a distributed setting where the commit data does not reside on the node performing validation, transactions must either send committed data to the server in the commit request, or they must engage in a multistage pro-

tocol to first validate the commit before sending the data. The first approach wastes network resources if the commit fails, while the second approach incurs latency and may stall other transactions if the commit succeeds. Active aborts, on the other hand, make it reasonable to use a single-stage protocol that sends committed data along with the commit request because of the low probability of a conflict.

Furthermore, asynchronous deprecations also permit better optimism and cache usage when retrieving blocks. A pessimistic alternative to the read protocol given above is to employ a cache update protocol that always contacts the server when requesting the latest version of a block instead of assuming the cached copy is up-to-date. This would ensure that a read/write transaction always operates on the latest copy of the data. However, doing so incurs a full network round-trip for every block read. With asynchronous deprecations, the transaction will know that it was overly optimistic and should therefore abort within half the time of a network round-trip of retrieving the block from the local cache. Thus, the only disadvantage of optimistically reading cached data is that events during the half round-trip window following the read could force the transaction to abort where the pessimistic read protocol would simply have blocked. However, we assume that write conflicts are sufficiently rare in the workload that the trade-off is worthwhile.

3.2.3 Read-Only Transactions

Read-only transactions operate in a model similar to snapshot isolation [7]. A read-only transaction is assigned a timestamp as soon as it starts and always reads versions from that timestamp. Choosing a particular timestamp to operate at ensures the global serializability of read-only transactions with respect to read/write transactions. Since read-only transactions do not have observable effects, they are always serializable with respect to each other.

Choosing an appropriate timestamp is critical both to ensuring the properties laid out in Section 3.1 and for the efficiency of read-only transactions. The more flexibility afforded in the assignment of a timestamp, the better locally cached data can be taken advantage of.

In order to guarantee local causality, the timestamp must be greater than the largest timestamp assigned to any committed local read/write transaction. Because the read/write protocol assigns timestamps in increasing order, it is sufficient to simply use a timestamp greater than that of the last locally committed read/write transaction. Note that, because read-only transactions do not have effects, multiple read-only transactions can be assigned the same timestamp.

The write protocol presented in the previous section ensures that updates to the server's block store always occur with monotonically increasing timestamps. This means that each client can keep track of a *global least upper bound* (GLUB)—a timestamp before which no further updates can occur—by keeping track of the timestamp of the last deprecation message or local commit. Because monotonicity guarantees that read/write transactions cannot be assigned a timestamp less than the GLUB, assigning a read-only transaction a timestamp less than or equal to the GLUB is sufficient to ensure conflict-freeness between read-only transactions and read/write transactions.

Furthermore, because local writes update the GLUB, the window of allowable timestamps always contains at least one timestamp, read-only transactions never have to block until another transaction widens the window.

Freshness. While assigning timestamps less than or equal to the GLUB is *sufficient* to guarantee conflict-freeness, it is not *necessary*. In particular, if the last GLUB update occurred more than ϵ seconds ago, this may be too restrictive to ensure freshness. Thus, each client keeps track of the *wall-clock* time of the last update to its GLUB. When a transaction with ϵ freshness

is started, if the GLUB is more than ϵ seconds old, the client requests a GLUB update from the server before assigning the transaction a timestamp. On a relatively active client with a large cache, such updates may not be necessary very often because of frequent GLUB updates from other communication with the server.

Assuming the network latency is less than ϵ, assigning the transaction a timestamp of the GLUB received from the server is sufficient to guarantee freshness. If the network latency is greater than ϵ, then it is impossible to simultaneously provide freshness and conflict-freeness in any system.

4 Implementation

Storage. The client and server share a versioned block storage system. This system provides support for versioned block lookups and retrievals, writes, and deprecations, and transparently provides aggressive in-memory caching. On the server side, it provides block durability, while the same mechanisms are used on the client side to perform aggressive on-disk caching. The storage system is completely unaware of the transaction protocol, instead providing hooks that are used by higher layers in the client and server to provide the protocol.

Communication. The client and server communicate via TCP using bi-directional asynchronous message passing built atop a custom channel multiplexing protocol and the Java serialization protocol. In order to reduce the space overheads incurred by Java serialization, we slightly modified the low-level protocol and hand-wrote serializers for messages with non-trivial state.

Client. The client library provides the transactional block store interface to the overall system. It provides methods to begin read-only and read/write transactions; to read, write, and create blocks; and to commit and abort transactions.

Server. The server responds to requests for blocks from clients, and commit requests. When a commit request is received, it performs a serial backwards-validation protocol to determine if the transaction can commit, and if so, commits it to stable storage and sends out deprecation messages to any clients that may have affected blocks in their cache.

5 Evaluation

We evaluated the system using a trace of NFS activity from the Berkley Auspex file system [10]. We compared conventional optimistic concurrency control to our read-optimized OCC algorithm for a number of performance metrics: network traffic, and client and server CPU usage. In each case, we found that our system offered substantial improvements over OCC for the file system workload. While both algorithms provide global serializability, conventional OCC incurs the additional cost of providing global causality, which this experiment quantifies.

The trace runner constructs a simple file system atop the block store and simulates the contents of the trace. For each host in the trace, the trace runner constructs an independent instance of the client library, complete with its own cache. The trace is replayed by simulating an ext2-like file system complete with fixed-size data blocks, inodes, indirect inodes and doubly indirect inodes.

Since the workload is not transactional, transactions were inferred from open and close requests. Stat requests were assumed to be single operation transactions. Another option for inferring transactions would have been to assume one transaction per file system operation. This assumption more closely emulates POSIX semantics, but does not exercise the system as well because the transactions are much shorter.

The trace runner simulates conventional OCC

by strictly using read/write transactions, even for transactions that perform only reads. In a pure read/write workload, our algorithm reduces to conventional OCC, with the optimization of active aborts.

Table 1 presents the performance of our system for 100,000 file system operations, spanning approximately 1 hour of the trace and involving 134 distinct clients. Transaction inference yielded 65883 read-only transactions and 3882 read/write transactions. In all situations, about 100 transactions were aborted due to conflicts.

Read-optimized OCC performs better all around because a large percentage of the file system workload is read-only. Network traffic is reduced because read-only transactions do not need to communicate with the server. Aggregate CPU usage is reduced because less overhead is incurred validating and serializing read-only transactions.

Unfortunately, because the Auspex traces were gathered by snooping NFS traffic, they cannot account for the effects of client-side NFS caching. This reduces the effectiveness of our own system's caching for the trace and thus the overall performance of read-optimized OCC. Therefore, for a complete file system trace, we would expect the performance benefits of our system over conventional OCC to be even more dramatic.

6 Related Work

Multiversion concurrency control. The general concept of multiversion concurrency control has a long history, dating back to Reed's work in 1978 [22, 23]. Many variations on its use have been proposed for use in both centralized and distributed database systems [8]. Multiversion concurrency is used most commonly in the form of snapshot isolation (also a weakened consistency model, as described below), which is implemented in several popular commercial databases. The benefit of multiversion concurrency is that having multiple versions of the same data object makes it possible to perform some operations without locking. The exact degree to which locks can be avoided depends on the specific protocol; variants range from multiversion two-phase locking, where both read and write locks are used, to optimistic timestamp-ordering, with no locks at all [9].

Optimistic concurrency control. Multiversion concurrency control is frequently combined with *optimistic* concurrency control [15], making it possible to avoid locks entirely by aborting and retrying if a conflict occurs. Our work uses optimistic concurrency control in essentially its standard form for read/write transactions. The use of a single server greatly simplifies validation and ordering in our optimistic concurrency control; a protocol like CLOCC [4] would be required for the multiple-server case. Though our mechanism for optimizing read-only actions could also be combined with a locking protocol for read/write transactions, we have chosen an optimistic protocol instead because previous work has shown that optimistic concurrency control provides a performance benefit in distributed systems [17].

Weak consistency. Many proposals for isolation levels less than full serializability exist [13, 6, 7, 5, 3]. The ANSI SQL standard [6] defines three such isolation levels, and many databases default to one of these. Snapshot isolation [7] is also used by many databases. Snapshot isolation ensures that each transaction reads from a consistent snapshot of the database, but is not fully serializable because it suffers from an anomaly known as *write skew* because it only detects conflicts between the write sets of two transactions, not read/write conflicts. Our handling of read-only transactions could be viewed as snapshot isolation, albeit perhaps on a less-than-current snapshot, but we avoid write skew by tracking both the read and write sets of read/write transactions.

The plethora of work attempting to categorize

	Conventional OCC	OCC with Read Optimization		
		$\epsilon = 2$ secs	$\epsilon = .1$ secs	$\epsilon = 0$ secs
Network traffic	34.8 MB	27.19 MB	28.14 MB	28.14 MB
Avg. client CPU time	.66 secs	.22 secs	.25 secs	.43 secs
Server CPU time	32.67 secs	9.55 secs	13.07 secs	25.63 secs

Table 1: Performance results for a trace of 100,000 file system operations. The trace was run with conventional OCC and read-optimized OCC with different freshness requirements, ϵ.

these weak consistency models attests to their complexity: for example, [5] is a critique of the definitions in [7], which is itself a critique of the definitions in the SQL standard [6]. It is similarly difficult to reason about the cases in which such consistency is sufficient. For example, snapshot isolation is generally considered "almost" serializable (to the extent that it is the highest level of isolation offered by many databases). Many workloads, such as the TPC-C benchmark, do indeed execute serializably under snapshot isolation; however, verifying that this is the case requires non-trivial analysis [11].

Our work is similar in spirit to these forms of weak consistency, having the same goal of improving performance by allowing some anomalies. However, motivated by the difficulty in reasoning about non-serializable execution levels, we relaxed the *causality* requirements instead of consistency. As a result, our protocol is not directly comparable to other weak consistency schemes; the only anomalies that can be observed are ones in which different, but still transactionally consistent, states are observed on two different nodes.

Read-only transaction optimization. A number of optimizations that take advantage of read-only transactions have been proposed. Garcia-Molina and Wiederhold defined consistency and currency requirements for read-only transactions, and introduced the concept of *insularity* to determine when a transaction can be executed locally on one node of a distributed database [12]. Our processing of read-only transactions is similar to the algorithms they propose, but our use of optimistic concurrency control for read/write transactions leads to some important differences. Other work (*e.g.* [28, 29]) has considered how to minimize the number of versions that need to be retained in order to optimize read-only transactions, but it is generally focused on avoiding locking rather than taking advantage of cached data in a distributed system.

C-Store [26] applies the same concept in a different environment, running read-only queries in a data warehouse on an old version using snapshot isolation.

7 Future Work

7.1 Multiple Servers

The protocol described in Section 3.2 assumes a single, central server; this decision was made to simplify the protocol and the implementation. With a few modifications, the protocol can also be used in a more distributed setting. We sketch an outline of such a modified protocol. For simplicity, we consider an environment where blocks are statically partitioned between multiple servers. If all blocks in a transaction are on one server, the transaction proceeds as before. Otherwise, one server is chosen as a coordinator, and it executes a two-phase commit protocol.

The client must ensure that messages are processed in timestamp order. With a single server, and an order-preserving (TCP) connection, this task is trivial, but with multiple servers it is somewhat complex. A simple solution is to maintain loosely synchronized clocks on all machines (using NTP or a similar protocol), then include

timestamps in each message and delay the processing of each message until its timestamp has been reached [4]; another alternative is to explicitly use logical clocks [16].

An open problem is that two-phase commit is necessary for multi-server consistency, but is at odds with the conflict-freeness of read-only transactions. Specifically, the problem is that when subordinates are in PREPARED state, the fate of the prepared transaction is unknown; if a read-only transaction attempts to retrieve an affected block, it must block until the transaction commits or aborts. One solution is simply to relax the conflict-freeness requirement, but this is undesirable. An alternative might be to assign transaction timestamps lazily: rather than having the client choose a timestamp, it could choose a range of timestamps to send to the server in its GET requests, and the server could choose a timestamp that avoids blocking on prepared transactions.

7.2 Generalized Causality

Out-of-band communication between nodes can expose the lack of global causality in the system. This could arise either from applications opening their own network connections, or through external observers with the ability to observe more than one node in the system. While little can be done about external observers except to reduce the acausality to an acceptable minimum, it is unfortunate that a distributed application wishing to use a transactional infrastructure service based on our algorithm cannot better characterize its causality needs to the infrastructure. Generalized causality would permit such characterizations so that applications would not be limited to the strict dichotomy between local causality and global acausality.

7.3 Dynamic Read-Only Timestamps

Our protocol currently assigns read-only transactions a timestamp no earlier than the GLUB, even if the freshness requirement ϵ is much more flexible. In some circumstances, it can be desirable to use an earlier timestamp for which more cached data is present. Choosing the best timestamp to maximize cached data availability is a difficult problem. The timestamp is currently assigned when a transaction begins; at this time no information is available about what data will be read.

We could instead assign timestamps lazily, initially beginning with a window of maximum allowable size (ending at the current time, and beginning at the timestamp of the freshness requirement or the last committed read/write transaction). Each successive read would choose an appropriate version, from the cache if possible, and *narrow* the window to its intersection with the validity interval of that version. This allows more flexibility to use cached versions. Additionally, hints about the read set of the transaction could be used to make a more informed decision at transaction start time.

7.4 Applications

We would like to evaluate the performance of our system using real applications in addition to traces. We would like to have some sort of database workload, perhaps using our system as the bottom layer of SimpleDB. A transactional file system would also make an interesting workload; an actual implementation would make it possible to avoid the NFS caching effects we observed on our trace workloads. A file system implementation might benefit from expanding the block store interface to be aware of file metadata, allowing for several optimizations; to be practical, it would also require a system for inferring transactions from existing non-transactional application workloads.

8 Conclusion

Systems that use distributed transactions, such as distributed databases and transactional file systems, often sacrifice full serializability for

performance. Unfortunately, this optimization makes it extremely difficult to reason about the correctness of the system. In this paper, we argued that similar performance benefits can be attained by weakening *causality* in lieu of abandoning serializability, thus maintaining the correctness of the system.

Our system provides a global serial ordering of transactions, but weakens causality by allowing read-only transactions to run on stale data. The stale data is guaranteed to be no older than some user-defined ϵ before the beginning of the transaction. Local causality is maintained by ensuring read-only transactions observe all previous, locally committed read/write transactions. Finally, our system ensures that read-only transactions never abort, conflict, or block, yielding vastly improved performance for workloads that are heavily read-only.

We implemented our system using an OCC-like protocol with active aborts for read/write transactions, and our optimized protocol for read-only transactions. We compared our system's performance to conventional OCC, which provides both global serializability and global causality. Our experiments using the Auspex traces showed that our system does indeed significantly out-perform conventional OCC without having to sacrifice global serializability.

References

[1] ACM. *The 1995 ACM SIGMOD International Conference on Management of Data*, San Jose, CA, USA, June 1995.

[2] ACM. *The 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, Kiawah Island, SC, USA, Dec. 1999.

[3] A. Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, Mar. 1999.

[4] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data* [1].

[5] A. Adya, B. Liskov, and P. O'Neil. Generalized isolation level definitions. In *Proceedings of the 16th IEEE International Conference on Data Engineering (ICDE '00)*, San Diego, CA, USA, Mar. 2000. IEEE.

[6] American National Standards Intitute. Database language - SQL. American National Standard for Information Systems X3.135-1992, Nov. 1992.

[7] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data* [1].

[8] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185 – 221, June 1981.

[9] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery In Database Systems*. Addison-Wesley, Boston, MA, USA, Feb. 1987.

[10] M. Blaze. NFS tracing by passive network monitoring. In *Proceedings of the 1992 USENIX Winter Technical Conference*, San Francisco, CA, USA, Jan. 1992. USENIX.

[11] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Transactions on Database Systems*, 30(2):492–528, June 2005.

[12] H. Garcia-Molina and G. Wiederhold. Read-only transactions in a distributed database. *ACM Computing Surveys*, 7(2):209–234, June 1982.

[13] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Granularity of locks and degrees of consistency in a shared data base. In *IFIP Working Conference on Modelling of Data Base Management Systems*, pages 1–29, 1977.

[14] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *Proceedings of the 1994 USENIX Winter Technical Conference*, San Francisco, CA, USA, Jan. 1994. USENIX.

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

[16] L. Lamport. Time, clocks, and ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[17] B. Liskov, M. Castro, L. Shrira, and A. Adya. Providing persistent objects in distributed systems. In *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP '99)*, Lisbon, Portugal, June 1999.

[18] B. Liskov and R. Rodrigues. Transactional file systems can be fast. In *Proceedings of the 11th ACM SIGOPS European Workshop*, Leuven, Belgium, Sept. 2004. ACM.

[19] C.-H. Moh and B. Liskov. TimeLine: A high performance archive for a distributed object store. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI '04)*, San Francisco, CA, USA, Mar. 2004. USENIX.

[20] J. K. Ousterhout, H. D. Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson. A trace-driven analysis of the UNIX 4.2BSD file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, Pacific Grove, CA, USA, Oct. 1991. ACM.

[21] G. Özsoyoğlu and R. T. Snodgrass. Temporal and real-time databases: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 7(4):513–532, Aug. 1995.

[22] D. P. Reed. *Naming and synchronization in a decentralized computer system.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, Sept. 1978.

[23] D. P. Reed. Implementing atomic actions on decentralized data. *ACM Transactions on Computer Systems*, 1(1):3–23, Feb. 1983.

[24] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir. Deciding when to forget in the Elephant file systemq. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)* [2].

[25] M. Stonebraker. The design of the POSTGRES storage system. In *Proceedings of the 13th International Conference on Very Large Data Bases (VLDB '87)*, Brighton, United Kingdom, Sept. 1987.

[26] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-Store: A column-oriented DBMS. In *Proceedings of the 31th International Conference on Very Large Data Bases (VLDB '05)*, Trondheim, Norway, Sept. 2005.

[27] W. Vogels. File system usage in Windows NT 4.0. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)* [2].

[28] W. E. Weihl. Distributed version management for read-only actions. In *Proceedings of the 4th ACM Symposium on Principles of Distributed Computing (PODC '85)*, Minaki, Ontario, Canada, Aug. 1985. ACM.

[29] K.-L. Wu, P. S. Yu, and M.-S. Chen. Dynamic finite versioning: An effective versioning approach to concurrent transaction and query processing. In *Proceedings of the 9th IEEE International Conference on Data Engineering (ICDE '93)*, Vienna, Austria, Apr. 1993. IEEE.