Mixing Consistency in Geodistributed Transactions: Technical Report

Mae P. Milano Andrew C. Myers

Cornell University, Ithaca, NY, USA

milano@cs.cornell.edu andru@cs.cornell.edu

Abstract

Weakly consistent data stores have become popular because they enable highly available, scalable distributed applications. However, some data needs strong consistency. For applications that mix accesses to strongly and weakly consistent data, programmers must currently choose between bad performance and possible data corruption. We instead introduce a safe mixed-consistency programming model in which programmers choose the consistency level on a per-object basis. Further, they can use atomic, isolated transactions to access both strongly consistent (e.g., linearizable) data and weakly consistent (e.g., causally consistent) data within the same transaction. Compile-time checking ensures that mixing consistency levels is safe: the guarantees of each object's consistency level are enforced. Programmers avoid being locked into one consistency level; they can make an application-specific tradeoff between performance and consistency. We have implemented this programming model as part of a new system called MyriaStore. MyriaStore demonstrates that safe mixed consistency can be implemented on top of off-the-shelf data stores with their own native, distinct consistency guarantees. Our performance measurements demonstrate that significant performance improvements can be obtained for geodistributed applications that need strong consistency for some critical operations but that also need the high performance and low latency possible with causally consistent data.

1 Introduction

Atomic, serializable transactions offer programmers strong guarantees that enable simpler reasoning and simpler code. However, these transactions cannot be efficiently implemented for geographically replicated data, because the overhead of coordinating distant replicas is unacceptably high. Unfortunately, replication is essential for low-latency data access in applications with global scale. Consequently, there has been a proliferation of data stores offering weak consistency (e.g., MongoDB [23], Redis [4], Cassandra [15], and Riak [14]).

While each such system is well suited to certain tasks, many applications have state that benefits from the guarantees of strong consistency. For such applications, a commonly used strategy is to use weakly consistent storage systems only for the operations for which they are optimized, and to keep critical data on safer data stores that offer stronger guarantees. For example, Netflix uses an eventually consistent data store to track video playback position, but stores more sensitive information in a durable, strongly consistent data store [22]. Many other modern geodistributed applications have similarly come to rely on ad-hoc mixing of multiple databases and other data stores that make distinct assumptions about consistency, availability, and other core system properties.

Unfortunately, this common strategy has a significant cost. Programmers must reason not only about the individual guarantees of their backing data stores (already challenging), but also about the emergent behavior when data from these stores mixes. And if a programmer needs to update state on two different stores atomically, they're stuck; even if both stores fully support atomic transactions, there's no easy way to coordinate atomic actions across them.

In this paper, we contribute a new mixed-consistency programming model that simplifies the process of programming against multiple data stores with heterogeneous consistency guarantees. This programming model is implemented by our system MyriaStore. MyriaStore provides a small, unified API and expressive embedded transactions language against which application programmers can write multi-store applications with atomic transactions. Crucially, data stores can still expose specialized operations specific to that store directly to application programmers.

MyriaStore's core guarantee is one of safety. When application code interacts with strong objects (objects stored on strong data stores), MyriaStore provides the semantic guarantees of strong consistency; similarly, when code interacts with weak objects, MyriaStore preserves the low latency and high throughput possible with weak consistency. Application code can use both strong and weak operations and even mix them in the same transaction. Nevertheless, strong operations retain all the guarantees of strong consistency, while *all* operations retain the guarantees of weak consistency. And transactions are truly atomic: no operations inside a transaction become visible outside the transaction until all operations do.

MyriaStore enforces these properties with two new mechanisms. First, at compile time, its new transactions language uses information flow control to ensure that the consistency of strong data is never compromised. Second, at run time, a novel, lightweight dependency tracking system handles the interactions between data from different consistency levels. MyriaStore also leverages information flow analysis to automatically *split* transactions across the data stores being used, in a way that preserves atomicity and isolation of the transactions.

MyriaStore provides an API and transactions language for accessing data but it is not a data store. Instead, applications are built using multiple *existing* backing stores, each offering its own native consistency guarantees—which MyriaStore leverages. In fact, legacy applications can continue to operate unmodified in parallel with MyriaStore applications while using the same backing stores, with the same interfaces, guarantees and performance characteristics they have always enjoyed. Thus, existing data need not be moved or modified; using the same stores, MyriaStore applications gain the ability to execute mixed-consistency transactions with strong semantic guarantees.

2 Motivation

Modern applications need to operate on data at multiple levels of consistency. But simultaneously programming against both weak and strong data is currently both ad hoc and error-prone.

2.1 Strong and weak consistency models

A *consistency model* imposes constraints on the order in which a system's users can observe updates to system state; in other words, it says what values should be returned by `get`(x) when others are calling `put`(x,v).

Consistency models are usually defined in terms of the possible *schedules* they permit. A *schedule* is just a log of system state events; it records the start time and completion time of every `get()` and `put()` call. For example, the *relaxed consistency* model of C11 [3] only permits schedules in which each call to `get`(x) returns a value v supplied by some call to `put`(x,v).

Recent literature has featured a variety of "weak" or "eventual" consistency models [20, 8, 15, 23, 5] whose common idea is that if the system is left alone, every call to `get`(x) eventually returns the same value v, from some previous call `put`(x, v). Meanwhile, those same papers also refer to a notion of "strong" consistency; the details vary here too, but the basic idea is that any call to `get(`x`)` should return whatever was stored during the most recent call to `put()` on x.

What "strong" consistency means also varies in the literature. In this paper, when we refer to a "strong" consistency model in the context of operations, we are referring to linearizability [12]. When we speak of a "strong" consistency model in the context of transactions, we are referring to strict serializability [24], which is linearizability at the level of transactions. By a "weak" consistency model, we mean a model that provides fewer guarantees (allows more schedules) than linearizability.

A weak consistency model of particular interest is causal consistency as defined by Lloyd et al. [20]. Informally, causal consistency enforces observed orderings; after a client reads some value x, that client now sees only values for any variables y that are at least as new as the values of y read by the writer of x. Causal consistency rules out observations that violate causality, but it does not, in general, require that clients see the most up-to-date values of variables. This helps performance because variables can be replicated with low coordination between replicas.

2.2 A running example

As an example of an application that needs mixed consistency, suppose that we are trying to build a scalable group messaging service we'll call *Message Groups*.

This service allows users to join groups and to post messages to all members of any group they have joined. Low-latency communication is key to a good user experience, so application servers are deployed across the world, and key data is replicated across these servers.

Communication latency between these servers—even that arising from speed-of-light delays—means that it is infeasible to keep the replicated data strongly consistent. Fortunately, there is no need to enforce a global, total order on displayed messages. It is only necessary to respect causality, so that messages and responses to those messages appear in the correct order. Since causal consistency suffices for storing users' inboxes, this data is georeplicated onto servers providing causal consistency.

However, strong consistency is needed for other data of this application. The membership of users in various groups should be consistent worldwide, so that all servers agree on who is supposed to receive which messages. Therefore, the set of members of each group is placed at a data store supporting linearizable transactions, which ensure serializability and external consistency.

Causal consistency and linearizability are well-defined consistency levels, but trying to mix these levels in the same application can break the guarantees that both levels claim to offer.

2.3 Mixing consistency breaks strong consistency

Suppose we run a spam-fighting contest to advertise Message Groups. We divide all the users into two teams; team A needs to send all the spam they find to mailbox `a`, and team B needs to send all the spam they find to mailbox `b`. The first team to find 1,000,000 spam messages is declared the winner.

To implement this contest, we extend the existing transaction for delivering mail with a few lines of code:

```
// inside the mail delivery transaction
if (a.inbox.size() >= 1000000 &&
    b.inbox.size() < 1000000) {
  declare_winner(a);
} else
if (a.inbox.size() < 1000000 &&
    b.inbox.size() >= 1000000) {
  declare_winner(b);
}
```

Confident that our code is correct, we ship it off to the application servers, where it awaits the first team to hit 1,000,000 messages. To our surprise, the next day we discover that the code has not declared a single winner; rather, the winner keeps switching between a and b!

The problem is that to avoid slowing down the core functionality of message delivery, the data used in the guard condition (i.e., the inbox sizes) are and should be stored with causal consistency. But the function `declare_winner()` manipulates data with linearizable consistency; and since the guard is evaluated with weak consistency, nothing guarantees that `declare_winner()` is invoked only once. Function `declare_winner()` itself, existing in a purely linearizable regime, is not designed to deal with the potential for multiple re-executions. In fact, every single message receipt to either team could cause the winner to switch.

The essential mistake is that weak data (the inbox size) influences strong data (the declared winner). The code of `declare_winner()` must handle the potential errors from weak consistency, even if it does not itself access weak objects. If this strong code is not programmed defensively, the influence of weak data may invalidate assumptions made by strong code. Allowing weakly consistent data to influence the control flow of the program, even within a linearizable transaction, can effectively downgrade the isolation level of the entire transaction. MyriaStore detects this problem at compile time and disallows code like this in which weak data corrupts strong consistency.

2.4 The need for mixed-consistency transactions

The example of the previous subsection has a second problem; within a single transaction, we are attempting to manipulate data from separate, mutually-unaware datastores operating independent transaction mechanisms. Without the technology of MyriaStore, we have no reason to expect the result of this execution to be atomic, isolated, or even complete.

To see the pitfalls inherent to this naive mixture, consider how Message Groups might implement message delivery. We have a strongly consistent list of members named `members` to whom we want to deliver a message `post`. The programmer might naturally write this code:

```
for (member : members)
  member.inbox.insert(post);
```

However, this simple loop does not deal with concurrent modification to the data. What happens if the `members` list, or a certain user's inbox, changes during the loop? The set of users who receive the message might not be consistent with the expected set of recipients; the loop might accidentally post to someone, or might miss someone it was supposed to post to.

Clearly some form of concurrency control is needed. In the distributed setting, the usual solution would be to perform the loop inside an atomic transaction that isolates this code from concurrent modifications. Unfortunately, the list of members is stored at a stronger consistency level than each member's inbox, making this a *mixed-consistency transaction* that is not supported by any current system.

A simple-minded way to performing the loop in a transaction is to upgrade all user inboxes to the same strong store that is used for the members, but this incurs an unacceptable performance cost.

Alternatively, assuming we have a causal store such as Eiger [21] that supports atomic transactions, we could start transactions simultaneously on both the strong and causal stores. But the implicit interactions between these transactions could create bugs. For example, if another process updates the membership list while mail is being sent, this could cause the strong transaction to abort and roll back. Some users could then receive the same message twice. We might think to fix this problem by adding a "delivered" flag to each message, to be set when the message is sent. If the flag is strong, transaction rollback can reset its value and messages will still be sent twice. If the flag is causal, the programmer has to be careful to update it only at the end of the causal transaction, because causal updates will not be rolled back if the transaction retries, leading to lost messages.

As this example shows, even rather trivial code can require the implementer to reason very carefully about the interactions between different consistency models in the presence of the possible failures. The complexity of this reasoning can easily become overwhelming.

Therefore, we take a different approach: MyriaStore simplifies programmer reasoning by directly supporting mixed-consistency transactions that provide atomicity and isolation. With mixed-consistency transactions, the message-delivery example can be implemented efficiently by wrapping the two original two lines of code in a transaction block. When combined with MyriaStore's static consistency flow analysis, this mechanism provides strong isolation and atomicity guarantees without requiring communication among the underlying stores.

3 Programming Model

Because MyriaStore is intended to support a variety of underlying data stores, including key–value stores, databases, and file systems, it offers a single common API and embedded transaction language for accessing data stores. The safety of transactions expressed in its transaction language can then be checked in a uniform way for all data stores.

3.1 MyriaStore API

The API boils down data access to three common operations: creating data objects, retrieving object data, and modifying data objects. Data objects are accessed via *handles* that serve as capabilities for data access. How these component abstractions are implemented depends on the underlying store. For example, for a key–value store, the data objects are key–value pairs, whereas for a file system, the objects are files.

In MyriaStore, the object that permits access to data is a handle. In user code, handles are opaque, lightweight objects that support a `get()` and a `put()` operation. They are instance of class `Handle`, a wrapper type tagged with the consistency level at which data is stored, the access permissions of that data (read-only, write-only, or read-write), and the type of the data:

```
class Handle<Level, Access, Type> {
    Type get();
    void put(Type);
};
```

To get a handle for a specific object, one must request it from a `DataStore`. The abstract class `DataStore` provides a common interface that exposes the functionality of an underlying data store to client code. Its interface is narrow; all a store needs to provide is a way to create an object, a way to retrieve an object by name, a consistency level, a notion of time, and a way to begin a transaction:

```
class DataStore<Level level> {
    TransactionContext begin_transaction();
    Clock local_time(); //optional
    Handle<level,all,T> newObject(Name, T value);
    Handle<level,all,T> existingObject(Name);
};
class TransactionContext {
    bool commit();
    DataStore store();
};
```

This abstraction can be implemented in a variety of ways. For example, a file system can be used as a backing store by using file descriptors as handles and by implementing `get()` with `read()` and `put()` with `write()`.

MyriaStore only requires a data store to provide the `get()` and `put()` operations on its data objects, but existing data stores often natively provide additional operations that do not fit into the simple get/put model. Therefore, MyriaStore allows implementations of `DataStore` to export additional operations to clients. Additional operations are declared by the `DataStore` class like methods that can be safely called from within a transaction. For example, this declaration of a MyriaStore interface for a SQL database provides an additional operation named `PreparedStatement`, allowing clients to submit prepared statements to the database:

```
class SQLDataStore {
 ...
  class SQLHandle : public Handle<...> {...}
  void Operation(PreparedStatement) (SQLHandle h,
                                     string prepared) {
    connection.prepared(prepared,h.sqlID);
  }
};
```

3.2 Consistency as information flow

Recall the buggy promotional-campaign code from section 2.2. MyriaStore prevents writing such code by expressing transactions in an embedded transactions language called MTL (for MyriaStore Transactions Language). Using MTL, applications can perform gets, puts, and store-defined operations on MyriaStore object handles, with the familiar assurances of transactional semantics. Crucially, when those assurances are just not possible, as in the promotional campaign, MyriaStore produces a static compiler error.

The insight behind preventing code from violating consistency guarantees is to treat consistency as a form of *integrity* [2] and to use an integrity information-flow type system to outlaw bad programs.

In an information-flow type system [25], values are associated with a label drawn from a lattice. The key guarantee of these type systems is *noninterference* [9]: values associated with labels lower in the lattice cannot influence actions with labels higher in the lattice. In the case of MyriaStore, the lattice elements are consistency models, and values are labeled according to the `Level` annotation on `handles`. Once the promotional campaign pseudocode is annotated with such labels, the problem becomes apparent:

```
// inside the mail delivery transaction
if (a.inbox<weak>.size() >= 1000000 &&
    b.inbox<weak>.size() < 1000000) {
  declare_winner<strong>(a);
} else
if (a.inbox<weak>.size() < 1000000 &&
    b.inbox<weak>.size() >= 1000000) {
  declare_winner<strong>(b);
}
```

This transaction creates a banned information flow from the inbox size (weak) to the `declare_winner()` function, which must be strong. MTL catches such invalid flows statically and rejects the unsafe transaction.

It should be noted that the labels on the above example are written out for clarity, but as discussed in the next section, explicit labels are not required in MTL transactions.

3.3 MTL by example

MTL is an expressive embedded language for implementing mixed-consistency transactions, implemented in C++. We explain its features by walking through the actual MTL implementation of the Message Groups application (Figure 1).

```
struct post {
  std::string str;
}

struct user {
  remote_set<Level::causal,
             Handle<Level::causal, post> > inbox;

  static set<post> get_posts(Handle<user> _this) {
    TRANSACTION(
     let_remote(usr) = _this IN (
       let_remote(inbox) = $(usr,inbox) IN (
        return inbox)));
    }
};

using MemberList =
      RemoteList<user,
                 Level::strong,
                 Level::causal>;

struct room {
  MemberList members;

  void add_post(Handle<post> pst) {
    TRANSACTION(
      let(hd) = members IN (
        WHILE (isValid(hd)) DO (
          let_remote(hd_contents) = hd IN (
            let_remote(hd_value) = $(hd_contents,val) IN (
              do_op(Insert,$(hd_value,inbox),pst),
              hd = $(tmp,next)))));
    }

    void add_member(user usr) {
        TRANSACTION(
          members = MemberList(usr, members));
    }
};
```

Figure 1: MTL implementation of Message Groups

Much of this code should look familiar to a C++ programmer; outside transaction blocks, it merely declares structs that contain library types, including two MTL library types (`remote_set` and `RemoteList`).

At the heart of MTL are transaction blocks, signified by the syntax `TRANSACTION(...)`. MTL transactions are expressive—for example, they can contain loops—but they provide the standard guarantees of atomicity and isolation. In particular, all transaction effects become visible at once, and the transaction itself operates against a stable snapshot of the underlying storage systems.

Code inside transaction blocks is expressed using a syntax that is constrained by the implementation of MTL using C++ macros and templates. The syntax of MTL transactions is defined in figure 2, but its meaning is perhaps best explained using the example code.

Consider the `TRANSACTION(...)` block in function `get_posts`. Its purpose is simple: given a handle for a user object, it retrieves that user's inbox. It uses two MTL constructs: remote binding and field referencing. In MTL, the remote binding `let_remote(`x`) = `e` IN (`$stmt$`)` evaluates the expression e to obtain a handle,

```
Stmt → ...
        | Stmt, Stmt
        | let_remote (Id) = Expr IN (Stmt)
        | let (Id) = Expr IN (Stmt)
        | WHILE ( Expr) DO (Stmt)
        | IF ( Expr) THEN (Stmt)
        | Id = Expr
        | do_op(Operation, Expr*)
Expr → ...
        | Expr ⟨BinaryOp⟩ Expr
        | ⟨UnaryOp⟩ Expr
        | isValid(Expr)
        | $(Id,Id)
```

Figure 2: MTL syntax

dereferences it by fetching the associated object from the remote store, and then makes that object available in $stmt$ under the name x. It's analogous to the C++ statement `{T &x = *e; stmt}` except that e evaluates to a handle to a remote object, which is analogous to a C++ pointer. The syntax `$(usr,inbox)` simply means an access to field `inbox` in object `usr`, like writing `usr.inbox` in C++.

The transaction in function `add_post` is more interesting. It introduces four new MTL constructs: simple let-binding, assignment, `WHILE`-loops, and `do_op`, and one MTL library function: `isValid(e)`.

A let-binding `let(x) = e IN (stmt)` merely binds the result of the expression e to the name x in $stmt$. In contrast to `let_remote`, it does not dereference handles. It's analogous to the C++ statement `{T x = e; stmt}`.

The variables introduced by `let_remote` and `let` are mutable, and their values can be changed by assignment statements of the form `x = e`. If x was bound via `let`, the expression `x = e` simply replaces the value at x with the result of e, but if x was bound by `let_remote`, the result of e overwrites the previous value of x at the underlying data store.

Key to the expressive power of MTL is its support for loops. The loop `WHILE(e) DO (stmt)` behaves precisely as one would expect; at each iteration of the loop, the condition e is evaluated; if true, $stmt$ is re-executed.

Finally, the construct `do_op(`$Name$, *args* ...`)` allows a transaction to invoke a custom operation $Name$ that is provided by a data store. For example, the code of `add_post` uses the "Insert" operation of the underlying causal store to efficiently insert elements into a remote causal set. It is the job of that causal store to ensure that Insert operations from different clients are merged with causal consistency.

3.4 Statically checking consistency labels

As discussed earlier, consistency is enforced in MTL by statically checking information flow. Static checking is enforced using a largely standard type system for static information flow [25]. Figure 3 gives selected typing rules for MTL. Each expression is assigned a consistency label ℓ that reflects the weakest consistency of any information used to compute the expression. The root of static checking is the consistency labels on MTL handles, which derive from their data stores. All other labels are automatically inferred from the code of the transaction itself. Updates to variables are only permitted when the new value has a consistency label at least as strong as the variable's current label.

A consistency label is also associated with the program counter within the transaction. This *program*

$$\vdash \mathit{Handle}\langle\ell\rangle : \ell \qquad \frac{\Gamma; pc \vdash e : \ell_1 \qquad \Gamma; pc : e' : \ell_2 \qquad \ell = \min(pc, \ell_1, \ell_2)}{\Gamma; pc \vdash e\ \langle \mathit{BinaryOp}\rangle\ e' : \ell}$$

$$\frac{\Gamma; pc \vdash e : \ell_1 \qquad \ell = \min(pc, \ell_1)}{\Gamma; pc \vdash \$(e, f) : \ell} \qquad \frac{\forall e' \in (e...) : e' : \ell \qquad \min(pc, \ell) = \ell}{\Gamma; pc \vdash \texttt{do\_op}(Name, e...)}$$

$$\frac{\Gamma; pc \vdash e : \ell \qquad \min(pc, \ell) = pc}{\Gamma; pc \vdash id = e} \qquad \frac{\Gamma; pc \vdash e : pc \qquad \Gamma; pc \vdash \mathit{stmt}}{\Gamma; pc \vdash \texttt{WHILE}\ (e)\ \texttt{DO}\ (\mathit{stmt})}$$

Figure 3: Selected typing rules for MTL

counter label, determined by uses of control constructs such as `IF` and `WHILE`, keeps track of code whose execution depends on weakly consistent information. In the information-flow literature, such flows via program control are known as *implicit flows*. The example code of section 2.2 is a case where such implicit flows need to be controlled to prevent inconsistency. Within the contexts guarded by weakly consistent expressions, strongly consistent data may not be updated, as in the unsafe call to `declare_winner()` in the example code.

3.5 Transaction splitting

The strict information-flow discipline of MTL prevents all operations at consistency level l from depending on any operation at consistency lower than l. This independence implies that transactions can be partitioned into phases corresponding to the consistency level of their operations.

The most significant challenge in splitting a cross-store transaction is control-flow dependence across data stores. For example, a while loop with a strongly consistent guard can have a body that contains both strong and weak operations. In fact, such a loop can be found in figure 1 at lines 28–32. Given two consistency levels, the entire while loop must run twice; in the first execution, the loop executes all strong statements within it; on the second execution, only the weak statements are executed, potentially using results computed by the strong statements.

To understand the semantics of transaction splitting, we can imagine the following process. Before executing a transaction, we first allocate an empty buffer. During the strong pass of the transaction, whenever we encounter a causal statement, we simply copy that statement into the buffer. If that causal statement depends on a strong expression, we evaluate the strong expression and replace it with the result. Once the strong pass is complete, this buffer will contain a valid, purely causal transaction. We next evaluate this residual causal transaction, and commit both.

In this way, MyriaStore is able to preserve the expected semantics of interleaved execution. Another benefit of transaction splitting is that it allows data stores which do not permit dynamic transactions, such as H-store [13] and Eiger [21], to nonetheless participate in MyriaStore cross-store transactions.

4 Enforcing consistency across stores

Recall that section 2.2 presented example code containing two major kinds of errors. MTL's information-flow checking prevents the first kind, in which transaction consistency was unintentionally downgraded. What has not yet been explained is how prevent the second kind of error, by coordinating mixed-consistency transactions while respecting the consistency guarantees of the underlying data stores.

Indeed, even if one uses two distinct systems that provide the *same* consistency guarantee, the combined system may provide a much weaker guarantee, or even no guarantee at all. The problem of compositionality of consistency models has been well studied [12], but current state-of-the-art systems still rely on the end user to determine if a consistency model is compositional, with what it composes, and how to use it safely alongside models with which it does not compose [27]. With MyriaStore, we aim to remove this complexity; rather than have to guess at the semantics of cross-store interactions, users are offered a sound consistency guarantee derived from the consistency of the underlying stores.

4.1 Defining mixed consistency

To offer a consistency guarantee, we need to know what the semantics of cross-store transactions should be. What exactly is meant by *mixed consistency*?

As described earlier, we want to enforce causal consistency for all objects stored with at least causal guarantees, and we want to enforce linearizability for all objects stored with at least linearizable guarantees.

This idea generalizes; given a target consistency model and a set of operations, we can always *project* the operations onto the model by selecting all operations involving objects that are stored with at least the guarantees of the target model. We can then ask if the result of that projection is consistent with respect to the target model.

Using this idea, we define *mixed consistency*. We say that a sequence of operations O exhibits *mixed consistency* if, for all consistency models M, the projection of O onto M satisfies the requirements of M. With mixed consistency, we now have a concrete answer to the expected consistency of arbitrarily mixed transactions—even when working with mixed transactions over incomparable consistency models.[1]

To enforce these guarantees in practice, MyriaStore needs to build on guarantees offered by non-compositional consistency models. While there are many non-compositional consistency models available in the literature today, recent work has identified variants of causal consistency as providing the strongest possible consistency guarantees while still permitting disconnected operation [1]; indeed, causal consistency already implies many useful programming properties required by weaker models. For this reason, the MyriaStore protocol upgrades all cross-store operations to causal consistency when the weakest store supports causal consistency.

Further, we want transactions to be atomic, so their effects are either entirely visible or entirely invisible to other transactions.

4.2 Causal ordering violations

MyriaStore relies on each underlying data store to enforce guarantees on its own objects, but adds its own lightweight tracking mechanism for operations that reference objects across multiple data stores. Since the information flow checking prevents strong operations from depending on weak operations, linearizability is not threatened by mixed-consistency transactions. The key challenge is rather to ensure that causal consistency holds even when causal operations depend onleariable operations.

Consider the following example of a causal ordering violation that we would like MyriaStore to prevent. We have two clients C_1 and C_2 running a MyriaStore application, and both clients perform transactions against the same two data stores. One data store, S, stores its objects with linearizable consistency; the other data store, W, is widely replicated as servers W_1, W_2, etc. for high performance, but stores objects with only causal consistency.

Client C_1 starts executing first. To begin, C_1 creates the causal object o_w at the replica of W called W_1. In the next transaction, C_1 creates the object o_s at store S. Object o_s contains a reference to o_w.

[1] Consistency models form a lattice [27], so there is always some minimum consistency model onto which we can project all operations.

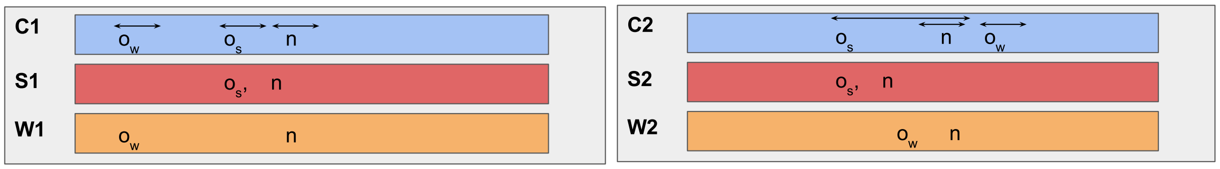


Figure 4: The gray boxes labeled with clients C_1 and C_2 contain three different timelines; the blue timeline corresponds to events as observed by the client, the red timeline indicates when data at S became available to the client, and the orange timeline indicates when data at W became available to the client. In this scenario, we observe that data at S becomes available to both clients simultaneously, while data written to W by C_1 can be delayed before it becomes available at W for C_2.

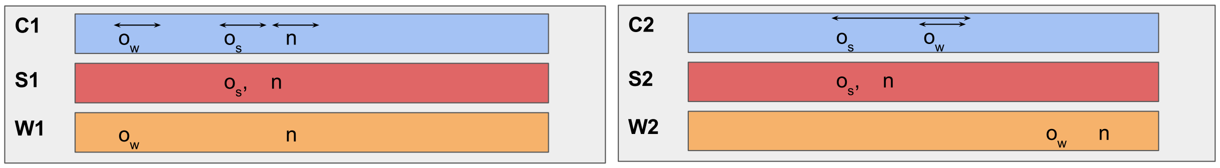


Figure 5: In this scenario, the tombstone does not become available for a very long time. Rather than wait, the client takes advantage of fetches a copy directly from a client that already has the necessary update.

Now client C_2 begins executing. It starts by reading o_s from S, and then tries to follow the reference to o_w at W. Unfortunately, the nearest replica W_2 has not yet learned of the creation of o_w, so C_2 cannot find o_w. This is a violation of causality; the creation of o_w preceded the creation of o_s, so causal consistency requires that o_w become visible no later than o_s.

4.3 Tombstones

To resolve such a violation of causality, there are two approaches: to slow down the propagation of o_s so that it arrives no earlier than o_w, or to speed up the propagation of o_w so that it arrives no later than o_s. In fact, MyriaStore takes both approaches, which we describe in turn.

Delaying causal reads with tombstones In the first approach, MyriaStore slows down o_s by blocking reads at C_2, as illustrated in figure 4. When C_1 attempts to perform a write operation on o_s at S, MyriaStore accompanies this operation with a *tombstone* value. This value is simply a unique, randomly generated nonce n. MyriaStore then additionally writes the tombstone to W. When C_2 subsequently attempts to read o_s, MyriaStore checks for an accompanying tombstone. When it finds that n accompanies o_s, MyriaStore knows that there might be some causal operation that has not yet reached C_2. It knows too that whoever wrote n to S must have also written n to W; by the guarantees of causal consistency, if C_2 can see n at W, it knows that every operation casually preceding n is also available at W. Thus, C_2 attempts to read n from W, blocking the read of o_s until n is found. Because W was trusted to guarantee causal consistency and the write of n to W happened after the write of o_w to W, once n is available, o_w will also be available.

Expediting causal propagation Slowing down an operation on its own is a sound but rather draconian way to resolve consistency violations. The second approach, therefore, is to speed up propagation of o_w. To do this, clients propagate causal updates without involving the causal store.

To allow this, we slightly alter the sequence of events discussed earlier, as shown in figure 5. When C_1 writes its tombstone, instead of simply writing a random value, MyriaStore writes the pair

$(nonce, ip\_address)$. C_1 additionally remembers the state of all values it has read up to this point, including the tombstone, in a *tracking set*.

When C_2 reads n, it finds this $(nonce, ip\_address)$ pair, giving it the information to contact C_1 directly. C_2 then contacts C_1 to retrieve the relevant *tracking set*. This set contains all the objects upon which o_s depends, so C_2 can safely proceed with reading o_s.

MyriaStore does not rely on this client-side propagation for correctness. This is important because C_1 may be unreachable, unwilling to cooperate, or untrustworthy. In these situations, C_2 simply has to wait for its local causal replica W_2 to catch up. However, client-side propagation seems quite applicable to the target domain, since clients that frequently share data are likely to be application servers running in the same trust domain, or even on the same network, so they have ready access to each other.

4.4 Making tombstones practical

While the protocol in the previous section is clearly sound, it raises some concerns for performance: linearizable writes become available only after the propagation delay of the underlying causal store, and the size of the tracking set used for client-side propagation can grow without bound. To address these concerns, there is a key observation: a client does not need to remember past versions of objects that are already available system-wide. In particular, if every client has seen all updates to some variable, there is no longer any cause to keep it in a *tracking set* or to retrieve it via client-side propagation. To exploit this observation, we need a mechanism to determine which objects are sufficiently old that everyone has heard about them; a global time, which everyone has reached, allowing garbage collection of updates from before that time.

It is an old result that tracking the dependencies on a set of k objects requires a vector clock with $k - 1$ entries; thus, stores which support causal consistency must somehow be tracking this information for all operations [16]. Indeed, this assumption is proven true in practice; causally consistent systems such as COPS, Eiger, 2-master PostgreSQL, and Bolt-on all either use these vector clocks directly or are easily modified to employ them [21, 20, 1].

To take advantage of this information, MyriaStore deploys a lightweight worker that periodically polls causal replicas and learns the most up-to-date object available at that replica. From all these collected vector clocks, the worker computes a new global-min time, by taking the minimum of each entry across all collected vector clocks. The global-min time will be as recent as possible while still being older than all collected clocks. The worker periodically broadcasts this time to all MyriaStore clients. Whenever a MyriaStore client receives a new minimum time, it discards all objects in its tracking set which are from before this time.

As the evaluation section shows, the tombstone protocol, with the global-min optimization, allows MyriaStore to perform well under heavy load from applications in the target domain.

4.5 Formal Algorithm Code

```
ONSTRONGWRITE
1  check tracking set
2  if tracking set non-empty
3     then update global-min
4          if tracking set still non-empty
5             then generate random nonce $n$
6                  write $(n, ip\_addr)$ to strong store
7                  write $n$ to causal store
```

```
OnCausalWrite
1  add object to *tracking set*
2  delete object from *remote-tracking set*, if present
```

```
OnCausalRead
1  **if** object present in *remote-tracking set*
2     **then if** object is older than version in *remote-tracking set*
3              **then** Add version from *remote-tracking set* to *tracking* set
4                   Return version in *remote-tracking* set
5              **else**
6                   Delete object from *remote-tracking set*
7                   add retrieved version to *tracking set*
8     **else** add object to *tracking set*
```

```
OnStrongRead
1  Check if $(n, ip\_addr)$ exists for this object
2  **if** so
3     **then** check if $n$ is available at causal store or is *remembered*
4            **while** $n$ is not available or remembered
5                **do** request *remote-tracking set* from $ip\_addr$
6                   **if** *remote-tracking set* received
7                     **then** *remember* $n$
8                          **break**
```

```
OnUpdateGlobalMin
1  remote objects from *tracking set* if behind global min
```

```
Periodically
1  update global-min
2  **for** each remembered $n$
3       **do if** $n$ is available at causal store
4              **then** remove $n$ from *remembered*
```

5 Implementation

MyriaStore has been implemented as five separate C++14 components: the core library, MTL, the tracking mechanism, support utilities, and the SQL datastore interfaces. The core library of MyriaStore, covering Handle interfaces, Datastore interfaces, and Operations, consists of approximately 2,300 lines of C++14 code, while MTL requires an additional 3,300. The tracking mechanism, which implements the algorithm from section 4, requires about 1,000 lines of c++14 code. The support utilities, including many compile-time template libraries, are about 3,500 lines of code.

Finally the SQL interface, implementing both a linearizable and causal interface to PostgreSQL, requires about 1,100 lines of code.

To simplify implementation, the current MyriaStore prototype supports working with up to two stores simultaneously in an MTL transaction. While applications involving more than two underlying data stores are increasingly the norm, many transactions in these multi-store systems only require transactions across objects in two stores, and thus would be a good match for MyriaStore even in its current prototype.

5.1 MTL

We implemented MTL as a DSL embedded into modern C++. The MTL language is written in pure C++14 templates, and can be compiled using any C++14-compliant compiler (for example, clang 3.7). We took this approach to follow the language-as-a-library paradigm; to use MTL in existing C++ projects, all one must do is `#include` the MyriaStore libraries. In order to achieve the semantics of variable bindings with type inference and information-flow enforcement in pure C++, MyriaStore makes heavy use of expression templates and introduces a novel scope-mangling technique by which we use CPP macros to convert let-binding statements into a pseudo-CPS format.

5.2 Implementing transaction splitting

Section 3.4 presented the semantics of transaction splitting intuitively, as being equivalent to copying code from strong executions to weak ones. This semantics did not correspond to a realistic implementation, however; the real implementation takes a different approach.

In order to cope with the interleaving of strong and weak operations, MyriaStore employs the well-known technique of static single-assignment, implemented via a novel mechanism called *indexed looping* [6]. Each syntactic occurrence of an expression in an MTL program is associated with two tags: a static tag indicating where in the MTL program the expression occurs, and a dynamic tag which counts the number of times that syntactic occurrence of the expression is accessed. Program execution is divided into one pass per consistency level. During the program pass for level l, when an expression or statement is encountered, MTL first determines if the expression or statement should run at level l. If so, MTL creates a new dynamic tag for that expression (by incrementing the old dynamic tag if available), and associates the result of running this expression with the pair of its static tag and new dynamic tag. When MTL encounters an item which does not execute at l, it checks to determine when the item should execute. If the item should execute after l, MTL recursively checks its subexpressions, and eventually moves on. If the item should execute before l, MTL checks if this is the first time it has encountered this item in the current pass. If it is, MTL retrieves the value associated with that item at dynamic tag 0. If this is not the first time, MTL retrieves the value associated with that item at the dynamic tag one greater than the previous time MTL visited this item[2]. In order to carry out transaction splitting in a database which does not permit a client to inspect a transaction's state before commit, these indexed states would need to be created at the remote database and retrieved from the database at the end of each pass. The current prototype does not handle this use case.

6 Evaluation

We use the MyriaStore implementation to model the intended application domain: user-facing application servers that share one linearizable and one causally consistent underlying storage systems, where the application servers are geographically distant from the linearizable storage system. We believe this closely mirrors reality; weakly consistent storage servers can be relatively close to application servers because they are able to withstand high latencies during replication and can therefore be separated geographically; linearizable data stores are more often housed within a single datacenter, because latencies encountered during replication have an outsized impact on their overall performance.

In this setting, we explore several key questions regarding the performance of MyriaStore:

[2]it will never be the case that MyriaStore "runs out" of dynamic tags; to do so would imply that MyriaStore was executing a loop whose loop guard is encountered more times than some elements of its body. As information flow requires the loop guard to execute during the first phase of this loop's visitation, this would imply that MTL managed to accidentally skip certain statements in the loop during an earlier execution. This is impossible.

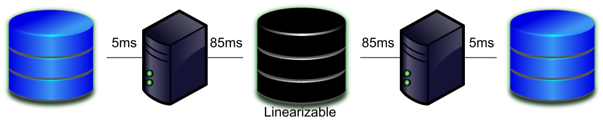


Figure 6: Experimental setup, including latencies between system elements

- Do mixed-consistency transactions, as promised, offer better performance than running similarly atomic transactions with strong consistency?
- What overhead is added by the tracking mechanism used to ensure that consistency guarantees are preserved when different consistency levels are combined?
- On what kinds of workloads does this mechanism work well or poorly? In particular, what is the performance impact of having different mixtures of strong and weak transactions?

6.1 Experimental setup

To test the performance of MyriaStore in practice, we created the configuration illustrated in figure 6. In this configuration, 250 logically separate application servers (represented by the black towers) each maintain connections to causally consistent (blue) and serializable (black) databases. Links between application servers and the serializable database experience a round-trip latency of 85ms±10ms; links between application servers and the causally consistent databases experience a round-trip latency of 5ms. Latency to the causal system was set by measuring ping times between an Internet2-connected university and its nearest datacenter; latency to the linearizable system was set by measuring ping times between Internet2-connected universities on the east and west coast of the United States. In both cases, latency simulations are provided by the netem kernel module on Linux 3.17. We employ 5 separate physical machines, each hosting 50 separate application server instances.

For driving load to application servers, we adopted a semi-open world model of events, with delay between events following the usual exponential distribution.

Using PostgreSQL as a backing store Our causal and linearizable data stores are both backed by PostgreSQL 9.4 running on dedicated machines. These PostgreSQL instances are configured with a maximum of 2010 connections, 128 MB of shared buffers, and with both `fsync` and `full_page_writes` disabled to improve performance; the rest of PostgreSQL's configuration parameters are left at their default values. These PostgreSQL instances consist of only two tables; one table associates integral values with integral keys and version numbers; the other table associates binary blobs with integral keys and version numbers. Any integral type is mapped to a row in the first table; all other types are mapped to a row in the second table. SQL queries over these tables are naive updates, selects, and increments (for the integral table). Because we use PostgreSQL as a key–value store, we do not believe that SQL-specific performance concerns such as query optimization or parse time have any significant impact on our results.

When running as a linearizable store, PostgreSQL is put in a "normal" operating mode with a single master per object and the `SERIALIZABLE` transaction isolation level. The coding overhead required to create this interface was pleasantly small; about 180 lines of C++ code, mostly for registering prepared statements.

Configuring PostgreSQL as a causally consistent store proved more challenging. Our approach is to run PostgreSQL via logical BDR (bi-directional replication), in which multiple PostgreSQL instances accept writes to the same tables and asynchronously propagate those updates to each other PostgreSQL instance. Each instance runs transactions at the "repeatable read" isolation level, corresponding to the guarantees of Snapshot Isolation. To enforce an ordering on operations occurring at distinct database instances, we use a vector clock as a version number for each row in these tables. A stored PL/pgSQL operation updates these version numbers upon row modification. All told, these mechanisms were implemented in 1,000 lines of C++ and about 100 lines of PL/pgSQL.

Snapshot Isolation in PostgreSQL in turn enforces the guarantees of causal consistency; within a single session, all reads will reflect data no older than the previous transactions' reads, and each transaction can see the modifications made by all previous transactions from this session. Finally, in order to enforce a notion of ordering for operations occurring at separate database instances, we associate a vector-clock as a version number for each row in these tables, using a stored PL/pgSQL operation to update these version numbers upon row modification. These vector-clock version numbers form a partial order consistent with causal consistency, and have been the basis for enforcing causal consistency in past work [cite bolt-on causal consistency]. Several application servers are associated with a single element of this vector; we say that all MyriaStore clients which share an entry in the vector-clock are in the same "causal group." All elements of the same causal group communicate with a single BDR instance to allow for safe use of their shared entry in the vector-clock.

In order to avoid merge conflicts during replication, each logically separate instance of the BDR database has an associated index value. Every object in each table is then stored once for each index value. Updates to a BDR replica include this index value, causing every BDR instance to write to a disjoint subset of the rows in each table. Reads are performed by reducing over all rows with the same key, regardless of their index value, to produce an aggregate response. For efficiency, this aggregation is carried out by a PL/pgSQL stored procedure and occurs automatically during replication.

6.2 Benchmarks

We could find no existing benchmarks for mixed-consistency transactional systems. To test MyriaStore, we developed two new benchmarks intended to represent the emerging mixed-consistency landscape. The first is a simple, fully synthetic benchmark that mixes read-only transactions that reference single rows from the integral-types table, and read-write transactions that read and increment values in the integral-types table. Objects referenced during read operations are selected from a Zipf distribution over 400,000 names; objects referenced during write operations are selected from a uniform distribution over the same names. The second benchmark is the Message Groups example discussed in section 2.2. Its three main transactions appear verbatim in figure 1. In this benchmark, the inbox we check for messages is drawn from a ZipF distribution, and the group to which a message is posted is selected from a uniform distribution.

The synthetic benchmark has the attractive property that it offers several different tuning parameters that allow us to explore the space of workloads. Before using it, we assessed the value of measurements made on the synthetic benchmark, by comparing its results against the more complex Message Groups benchmark. When running the Message Groups benchmark, we considered each individual store operation to be an "event," and only considered configurations in which users repeatedly post the same message to groups of size one. This configuration was chosen to provide maximum control over which operations write, read, are strong, or are causal. When running the synthetic benchmark in a similar configuration, we find that results from the synthetic benchmark (figure 7) closely match the performance results from Message Groups, as desired.

6.3 Results

Speedup relative to strong consistency The most important question for MyriaStore performance is whether mixed-consistency transactions offer a speedup compared to the simple alternative of running transactions entirely with strong consistency. As shown in figure 8, we compared the maximum achievable throughput of these two configurations over a range of workloads. In part (a), we see that for identical

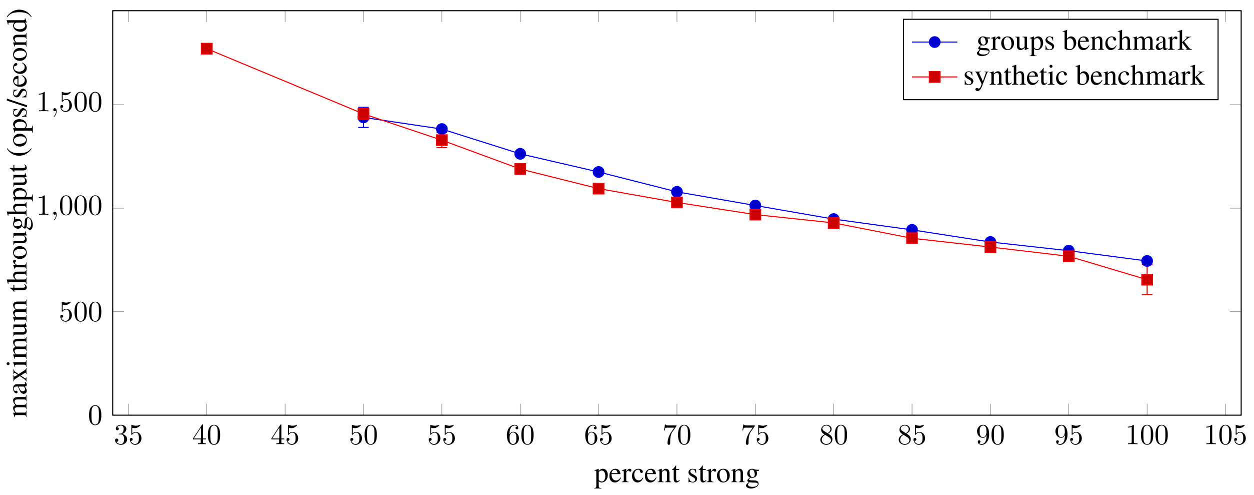


Figure 7: The Message Groups program and the synthetic benchmark running in similar configurations. In both, 95% of operations are reads and 25% of operations are strong. Specifically, 24% of the operations are group membership checks, 1% of the operations are group joins, 71% of the operations retrieve a message from an inbox, and 4% of the operations are message posts. Users always join the same group or post the same message, so group and inbox size do not grow with time.

workloads, the maximum throughput is improved by making even a small ($\tilde{2}0\%$) fraction of transaction operations causally consistent – and as the share of causal operations grows, the performance improves. Additionally, we observe that the speedup obtained by using the close, causal stores is roughly linear in the share of causal transactions. This result is in line with expectations: as fewer transactions incur high latency and serializable ordering, performance should increase.

Below 40% strong, MyriaStore's total maximum throughput always measures above 2,500 operations/second; thus, the observed improvement is not simply a matter of available hardware resources.

Overhead of tracking Now that we have established that mixing causal and serializable operations increases maximum throughput, we turn to the overheads incurred by the tracking algorithm as presented in section 4.3. As seen in 8(a), the tracking mechanism has a noticeable, constant effect on the maximum throughput of the system. The lower line in this figure should be read as a baseline for PostgreSQL itself; there are no MyriaStore functions running during this test, with the MyriaStore API instead directly translated into prepared SQL statements. The difference in maximum throughput has a simple cause; as noted in the discussion of the tracking algorithm, every attempt to read from a strong store must be accompanied by an additional search for a "tombstone," which would inform us that the strong object in question carries causal dependencies. Though these tombstones are rare, the check itself increases load on the system.

Latency The tracking mechanism does have an effect on latency, especially because of design decisions made for backward compatibility. Figure 9 shows the effect of the tracking mechanism on observed latencies. Latencies are presented as a CDF collected from the system running at 80% of maximum throughput, in a configuration in which where 95% of all operations are reads, 75% of all operations use the causal store, and 25% of all operations use the strong store. We ran this test twice, once with the tracking system enabled, and once without. Figure 9 (a) focuses on operations which only use the causal store. The red line on this graph represents the data collected without the tracker running; the green and blue lines represent writes and reads collected with the tracker running. As expected (section 4.3), the tracking mechanism adds essentially

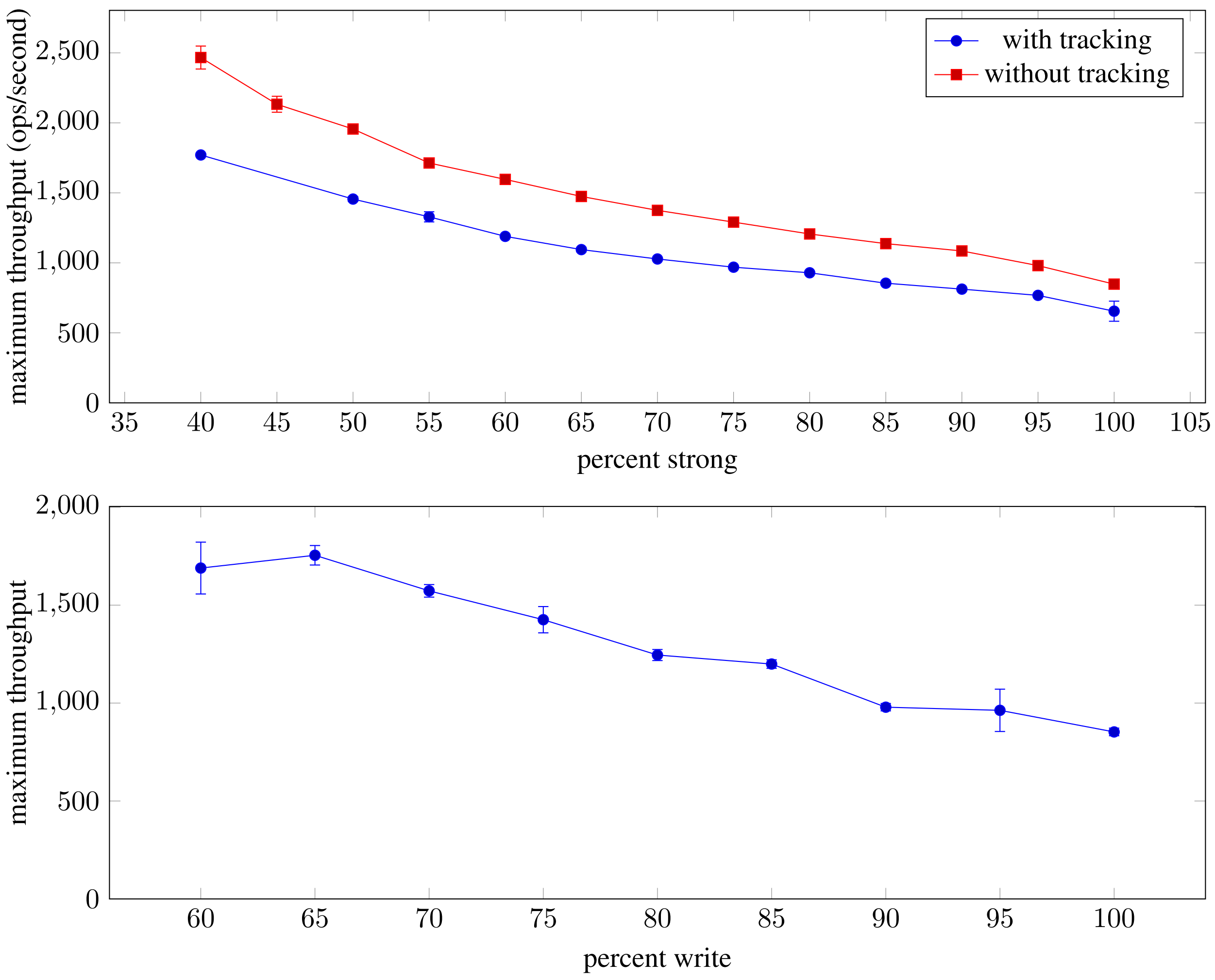
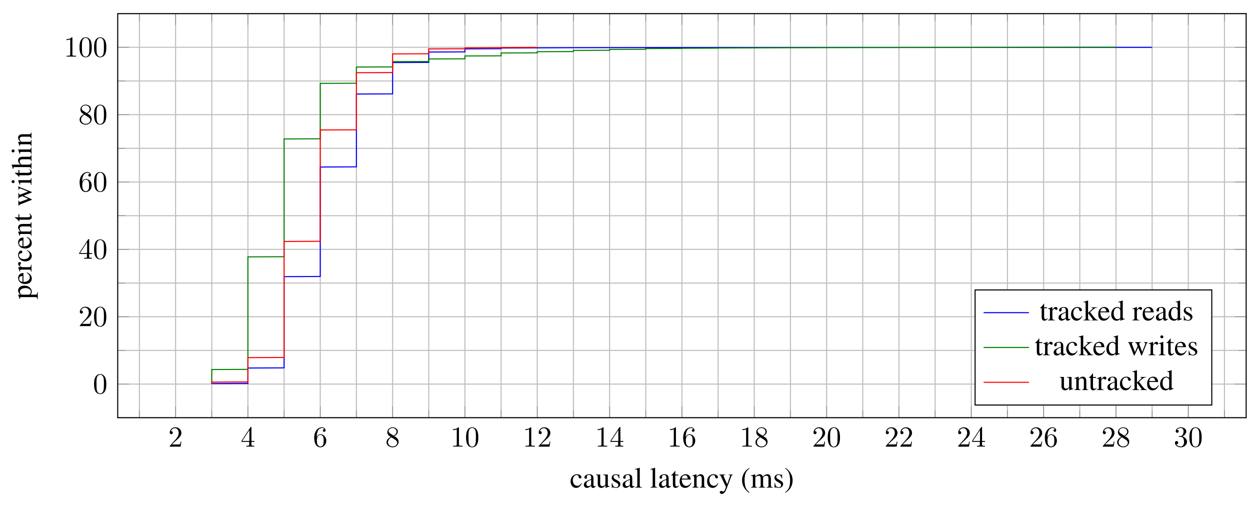


Figure 8: (a) maximum throughput as a function of strong mix for a 95% read workload. The red series shows maximum achievable throughput without tracking; the blue series shows full tracking. (b) Maximum throughput as a function of write share for a 75% causal workload.

no overhead to causal operations.

Figure 9(b) reports latencies for operations that utilize the strong store. Again, the red line represents all strong operations running without tracking, the green line represents all strong writes running with tracking enabled, and the blue line represents all strong reads running with tracking enabled. One might be surprised that with the tracker, strong read operations are slower than strong write operations. This outcome is, however, to be expected; as presented in Section 4.3, the tracker must check for a tombstone on *every* strong read, while it only writes a tombstone on a few strong writes. In fact, if one looks closely at the green line, one can observe the $<5\%$ of writes which do need to write a tombstone.

The latency results are a motivation to speed up strong reads so that their overhead is comparable to that on strong writes. There are two obvious (but still unimplemented) approaches: first, bundling tombstones with their strong objects, and second, parallelizing tombstone reads. We expect that the first approach would improve strong read performance by a factor of two, but it does comes at a price: in order to bundle a tombstone with the strong object itself, we would need to change the representation of data on the store, breaking any legacy clients attempting to use the data. The second idea comes with no such caveats; fetching



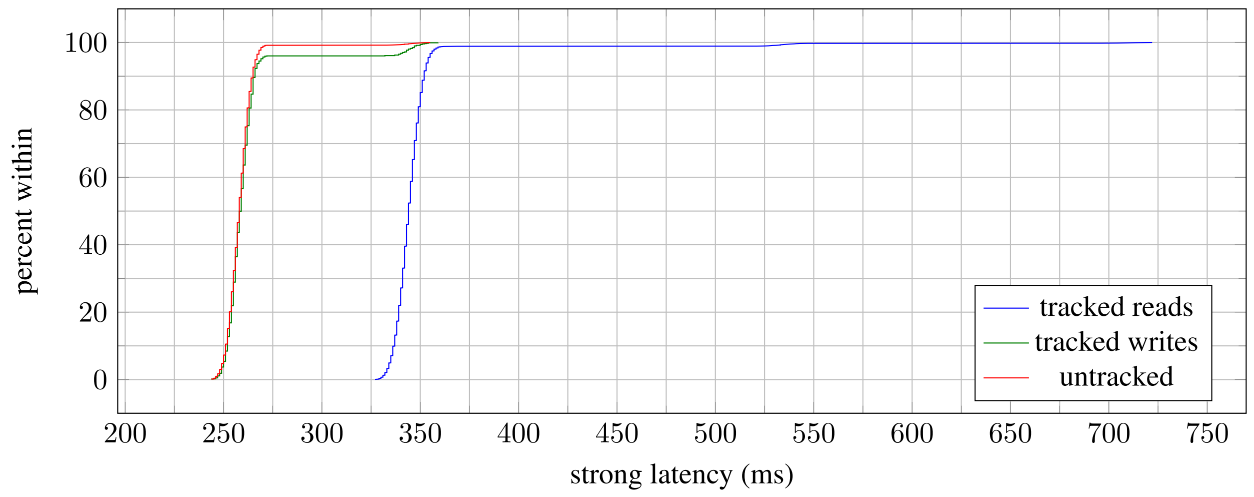


Figure 9: CDF plots for operation latency. (a) shows latencies for causal operations without the tracker (red), and breaks out writes (green) and reads (blue) for latencies with the tracker. (b) shows latencies for strong operations without the tracker (red), and breaks out writes (green) and reads (blue) for latencies with the tracker.

the tombstone in parallel with the object itself would work just fine.

Figure 9 shows that a small fraction of strong reads experience much higher latency than normal, over 500ms. This effect results from the algorithm of Section 4.3. These operations encountered a tombstone upon strong read, but were unable to find a matching tombstone in their nearby causal replica. This means some updates were missed, so they attempt to fetch the missing updates directly from a different client while, in parallel, rechecking a nearby causal replica for the missing tombstone. Eventually one of these operations succeeds, and the strong read is allowed to complete. These operations are the only place where cooperative caching is used, accounting for less than 0.25% of all strong reads.

7 Related Work

7.1 Mixing consistency on a single system

Most work in the mixed-consistency space has focused on operations against a single distributed system. Some systems [27, 18, 17, 11], and even SQL provide users with a language of constraints which they can use to describe the invariants associated with their data and/or operations and in turn provide the weakest consistency possible while still satisfying those constraints. These languages range in complexity from simple annotations [18] to entire programming languages [27].

Much other single-store work has focused on providing single data stores which supply multiple consistency levels, allowing users explicitly choose the consistency level associated with each operation or object [23, 5, 15, 8, 18]. While this certainly allows programmers a level of control over the safety and speed of their operations, these works make no guarantees about the interactions of their various consistency and isolation models, making the job of programming against these systems just as difficult as programming against multiple, single-consistency distributed stores. Indeed, MyriaStore chooses to treat such stores as semantically separate distributed systems for the purposes of safety. A smaller set of work has focused on providing similar tools for isolation levels across transactions [29].

Still more work focuses on automatically adjusting the consistency of operations without the need for programmer annotations at all. These systems range from the actively unsafe [28] to the semantically linearizable [19], with many in between [17, 30]. While this approach is significantly easier for the programmer to understand, it can often admit strange performance characteristics. For example, changing a single transaction somewhere in the system can significantly affect the performance of unrelated transactions [18, 30, 26].

This laudable work lies somewhat outside the scope of MyriaStore's goals; we envision each of these systems as one among many stores a programmer could use in a single MyriaStore application. Indeed, the MyriaStore API is sufficiently minimal that these systems can expose any annotation-based functionality directly to the user, allowing a seamless integration of the safety provided by these systems alongside the cross-store safety and programmability made possible by MyriaStore.

7.2 Consistency across multiple systems

A much smaller body of work attempts to make the job of programming against multiple data stores easier. The most obvious candidate is SQL, and the SQL compatibility libraries like JDBC and ODBC [10]. These standardized languages attempt to provide a unified API for programming against every RDBMS; and while each different database has its own unique implementation of the SQL standard, much of the language is shared, making it easy to port simple code which ran against one RDBMS to run against a different one. The SQL language itself is only aware of a single database system, leaving the work of coordinating actions across multiple database systems up to the programmer. Additionally, issues of consistency and isolation level are not addressed in SQL itself; rather, each underlying system determines which actions are safe.

7.3 Enhancing the consistency of existing systems

Beyond SQL, some existing work has focused on mechanisms which can upgrade the consistency guarantees of weakly-consistent underlying stores [1]. Indeed, several previously-discussed projects [21, 27] use this approach internally, adding consistency layers atop existing distributed systems like Cassandra.

7.4 Enhancing performance of distributed systems

The mechanism for expediting propagation of causal updates can be seen as a form of cooperative caching [7], though it is updates that are shared among clients rather than object states.

8 Conclusion

We have introduced a new programming model for writing modern geodistributed applications that need to trade off performance and consistency. The mixed-consistency transaction model offered by MyriaStore makes it possible for programmers to safely combine strongly and weakly consistent data in the same application, with confidence that weakly consistent data does not corrupt the guarantees of strongly consistent data. Appealingly, this model can be implemented in a backward-compatible way on top of existing data stores that offer their own distinct consistency guarantees, without disrupting legacy applications that share the same stores. The performance results suggest that for geodistributed applications, mixed-consistency transaction allow programmers to achieve higher performance by using weakly consistent data in a selective way.

9 Acknowledgments

The authors would like to thank Josh Brown, Stephanie DeMane, Jon Sailor, Cody Mello, and Robert Mustacchi for their help determining industry applications of heterogeneous consistency. The authors would like to than Ethan Cecchetti for his help configuring PostgreSQL, and Nate Nystrom for discussions about information flow and consistency.

This research was conducted with Government support under and awarded by DoD, Air Force Office of Scientific Research, National Defense Science and Engineering Graduate (NDSEG) Fellowship, 32 CFR 168a. There's also a summer I spent working on this under the Fabric grant, so I should paste that in here too.

References

[1] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Bolt-on causal consistency. In *ACM SIGMOD International Conference on Management of Data*, 2013.

[2] K. J. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Bedford, MA, April 1977. (Also available through National Technical Information Service, Springfield Va., NTIS AD-A039324.).

[3] Hans-J Boehm and Sarita V. Adve. You don't know jack about shared variables or memory models. *Comm. of the ACM*, 55(2), 2012.

[4] Josiah L. Carlson. *Redis in Action*. Manning Publications Co., Greenwich, CT, USA, 2013.

[5] Brian F Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment*, 1(2):1277–1288, 2008.

[6] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. on Programming Languages and Systems*, 13(4):39, 1991.

[7] M. Dahlin, R. Wang, T. Anderson, and D. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Usenix Symposium on Operating Systems Design and Implementation*, 1994.

[8] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *21st ACM Symp. on Operating System Principles (SOSP)*, 2007.

[9] Joseph A. Goguen and Jose Meseguer. Security policies and security models. In *IEEE Symp. on Security and Privacy*, pages 11–20, April 1982.

[10] Graham Hamilton, Rick Cattell, Maydene Fisher, et al. *JDBC Database Access with Java*, volume 7. Addison Wesley, 1997.

[11] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages*, 1(13):124–149, Jan 1991.

[12] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. Technical Report CMU-CS-88-120, Carnegie Mellon University, Pittsburgh, Pa., 1988.

[13] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *PVLDB*, 1(2), August 2008.

[14] Rusty Klophaus. Riak core: Building distributed applications without shared state. In *ACM SIGPLAN Commercial Users of Functional Programming*, CUFP '10, pages 14:1–14:1, New York, NY, USA, 2010. ACM.

[15] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

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

[17] Cheng Li, Joao Leitão, Allen Clement, Nuno Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. Automating the choice of consistency levels in replicated systems. In *USENIX Annual Technical Conference*, 2014.

[18] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, ca Nuno Pregui and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *10th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, 2012.

[19] Jed Liu, Tom Magrino, Owen Arden, Michael D. George, and Andrew C. Myers. Warranties for faster strong consistency. In *11th USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, pages 513–517, April 2014.

[20] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *23rd ACM Symp. on Operating System Principles (SOSP)*, 2011.

[21] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Stronger semantics for low-latency geo-replicated storage. In *10th USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, pages 313–328, 2013.

[22] Sagar Loke and Christos Kalantzis. Introducing Raigad—an elasticsearch sidecar. http://techblog.netflix.com/2014/11/introducing-raigad-elasticsearch-sidecar.html, 2014.

[23] Peter Membrey, Eelco Plugge, and DUPTim Hawkins. *The definitive guide to MongoDB: the noSQL database for cloud and desktop computing*. Apress, 2010.

[24] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, October 1979.

[25] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[26] Dennis Shasha, Francois Llirbat, Eric Simon, and Patrick Valduriez. Transaction chopping: algorithms and performance studies. *ACM Trans. on Database Systems*, 20(3):325–363, September 1995.

[27] K. C. Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. Declarative programming over eventually consistent data stores. In *36th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, June 2015.

[28] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *24th ACM Symp. on Operating System Principles (SOSP)*, 2013.

[29] Chao Xie, Chunzhi Su, Manos Kapritsos, Yang Wang, Navid Yaghmazadeh, Lorenzo Alvisi, and Prince Mahajan. Salt: Combining ACID and BASE in a distributed database. In *11th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, volume 14, pages 495–509, 2014.

[30] Chao Xie, Chunzhi Su, Cody Littley, Lorenzo Alvisi, Manos Kapritsos, and Yang Wang. High-performance ACID via modular concurrency control. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 279–294, New York, NY, USA, 2015. ACM.