# A Language for Mixing Consistency in Geodistributed Transactions: Technical Report

MATTHEW MILANO, Dept. of Computer Science, Cornell University, United States of America
ANDREW C. MYERS, Dept. of Computer Science, Cornell University, United States of America

Programming concurrent, distributed systems is hard—especially when these systems mutate shared, persistent state replicated at geographic scale. To enable high availability and scalability, a new class of weakly consistent data stores has become popular. However, some data needs strong consistency. To manipulate both weakly and strongly consistent data in a single transaction, we introduce a new abstraction: mixed-consistency transactions, embodied in a new embedded language, MixT. Programmers explicitly associate consistency models with remote storage sites; each atomic, isolated transaction can access a mixture of data with different consistency models. Compile-time information-flow checking, applied to consistency models, ensures that these models are mixed safely and enables the compiler to automatically partition transactions. New run-time mechanisms ensure that consistency models can also be mixed safely, even when the data used by a transaction resides on separate, mutually unaware stores. Performance measurements show that despite their stronger guarantees, mixed-consistency transactions retain much of the speed of weak consistency, significantly outperforming traditional serializable transactions.

**ACM Reference Format:**
Matthew Milano and Andrew C. Myers. 2020. A Language for Mixing Consistency in Geodistributed Transactions: Technical Report. 1, 1 (November 2020), 30 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

Programmers often face the task of writing concurrent, distributed systems that share mutable, persistent state across geographic distances. Traditional tools, such as strictly serializable atomic database transactions and distributed locking, do not scale across continents; the speed of light simply isn't fast enough for the cross-continental round trips needed by traditional transactions.

New tools have evolved to fill the gap, relying on weaker consistency to enable lower latencies and higher availability. Evidence from both industry [9, 21, 33, 37, 48] and academia [14, 29, 54] suggests that weak consistency is viable, especially when accompanied by familiar transactional abstractions, clear consistency guarantees, and efficient operations over persistent data.

But problems remain. While weaker consistency models suffice for certain application data, other data needs stronger consistency—single applications might need multiple *levels* of consistency. Running with a single consistency level, as seen in prior systems [8, 16, 31, 59, 61, 62]), can be slow [6]; all operations within a transaction must be upgraded to the consistency required by the most sensitive among them, introducing unnecessary delay and contention for objects that only require weak consistency. This is a fundamental problem: we need a general way to construct transactions that access data at multiple consistency levels, without compromising strong consistency where it is needed.

These concerns lead us to a key observation: consistency is a property of information itself and not only of operations that use this information. Further, the consistency with which we manipulate

Authors' addresses: Matthew Milano, Dept. of Computer Science, Cornell University, Gates Hall, Ithaca, New York, United States of America, 14853, milano@cs.cornell.edu; Andrew C. Myers, Dept. of Computer Science, Cornell University, Gates Hall, Ithaca, New York, United States of America, 14853, andru@cs.cornell.edu.

© 2020 Association for Computing Machinery.
XXXX-XXXX/2020/11-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

information should always match or exceed the consistency at which we store it. Based on this observation, we introduce *mixed-consistency transactions*. Each mixed-consistency transaction can operate over any and all data, even if this data is stored with varying consistency guarantees. Despite this expressivity, we maintain the consistency guarantees required by each data object by preventing less-consistent information from influencing more-consistent data. Mixed-consistency transactions are atomic: no operations inside a transaction become visible outside the transaction until all operations do.

In implementing mixed-consistency transactions, we uncover a further complication: engineers at major companies frequently find themselves writing distributed programs that mix accesses to data from multiple existing storage systems, each with distinct consistency guarantees [7]. It is unrealistic to assume that data can be freely migrated into ever-newer and more capable storage systems, or that all applications can be written against a single unified system; we therefore want to operate against multiple backing stores within the same mixed-consistency transaction.

Combining these challenges, we present MixT, a domain-specific language for mixed-consistency transactions. In MixT, persistent data and operations at various stores can be accessed with strong guarantees (§3). To ensure the semantic guarantees of mixed-consistency transactions, weaker-consistency information should avoid influencing stronger-consistency information. To prevent this influence, we view consistency as a property of data, treating consistency as a form of data integrity [5] expressed as labels on types in the language. Static analysis of information flow [50] then ensures that consistency guarantees cannot be violated by exposure to objects with weaker consistency.

The MixT language implements mixed-consistency transactions using three novel mechanisms (§4–5):

- Compile-time information flow control ensures that the consistency of data is never weaker than the level described by its storage location.
- Using information flow analysis, the code of each transaction is automatically *split* into sub-transactions for each consistency level, while preserving atomicity.
- A lightweight run-time mechanism ensures transactional atomicity, even between sub-transactions executing on multiple mutually unaware backing stores.

MixT works on top of stores' existing transactional mechanisms, without changing the representation of existing data, allowing existing applications to operate unmodified alongside MixT applications. And MixT can be easily adapted to a new store, by inserting the store's consistency level into MixT's consistency lattice and providing bindings to custom data operations specific to that store.

As we show experimentally (§9), mixed-consistency transactions perform well. MixT enables significant speedup vs. serializable transactions by exploiting weak consistency, without losing the guarantees sacrificed by current systems when consistency levels mix.

## 2 MOTIVATION

### 2.1 A Running Example

Suppose we are building a scalable group messaging service called *Message Groups*. This service allows users to join groups and to post messages to all members of any group they have joined. For low-latency communication, application servers are deployed across the world, with data replicated across these servers.

Communication latency between these servers makes it difficult to keep the replicated data fully consistent without degrading user experience. Fortunately, there is no need to enforce a global, total order on displayed messages. It is only necessary to respect potential causality, so that messages

```
var iterator = users,
while (iterator.isValid()) {
  log.append(iterator->v.inbox.insert(post)),
  iterator = iterator->next
}
```

Fig. 1. Naive code for sending messages. Corrected MixT code is found in Figure 4.

precede their responses. We therefore geo-replicate user inboxes at a weaker consistency level, *causal+ consistency*, which respects causality but does not guarantee a total order [41].

However, other data in this application requires stronger consistency. 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 store supporting *linearizable transactions*, which ensure serializability and external consistency [13, 27, 47]. Latency to this single store is necessarily much higher for many users than latency to their own inboxes.

### 2.2 The Need For Mixed-Consistency Transactions

To see the pitfalls inherent to this naive mixing of consistency levels, consider how Message Groups might implement logged message delivery, using the code in Figure 1. There is a linearizable list of members named `users` to whom a message `post` should be sent. Each member in `users` has a causally consistent inbox. For maintenance purposes, the sending of messages is logged (via `log.append`). The log does not even require causal consistency; instead, we might replicate it at *eventual consistency* [58], which requires only that reads converge after a sufficiently long quiescence. All mutations, including append and insert, are replicated across continents.

However, the simple loop in Figure 1 does not address concurrent modification to the data. Suppose that a thread concurrently modifies `users` while another thread is executing Figure 1. Without care, this concurrent modification might invalidate Figure 1's iterator; at best, it is unclear whether the new member will receive a message. As written, there is no reason to expect the result of this execution to be atomic, isolated, or even complete.

Clearly, some form of concurrency control is needed. We might change over to a recent system such as Quelea [54] or Salt [59], which provide both fully atomic transactions and multiple consistency levels. But these systems can only execute a given transaction at a single consistency level. In these systems, all data in the Message Groups example would effectively be upgraded to linearizability. There would be no performance benefit from having a weakly consistent inbox and log; message delivery performance would likely be unacceptable.

Alternatively, we could partition our data onto three distinct systems, each optimized for the appropriate consistency level. If we had a causal store such as TaRDiS [14] that supports interactive atomic transactions, we could start a separate simultaneous transaction at each system. This would achieve the desired performance, but the implicit interactions between these transactions could create bugs. For example, if another process updated the membership list while mail was being sent, the linearizable transaction might abort and roll back, restarting the loop. Without code to explicitly roll back the other concurrent transactions—which may not even always be possible—some users could then receive the same message a second time. 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 linearizable, 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 might not be rolled back if the transaction retries, leading to lost messages. And even if

```
if (a.inbox.size() >= 1000000 &&
    b.inbox.size() < 1000000) { // weak condition
  a.declare_winner()            // strong effect
} else
if (a.inbox.size() < 1000000 &&
    b.inbox.size() >= 1000000) {
  b.declare_winner()
}
```

Fig. 2. Contest logic inside the mail delivery transaction. Corrected MixT code is found in Figure 15.

we were to carefully solve this interaction, there is still the matter of the logger, a component that requests even weaker consistency (and expects correspondingly faster access).

Thus, this transaction cannot be naturally implemented using each underlying store's mechanisms in isolation. It requires a new form of *mixed-consistency*, *mixed-location* transaction not supported by any existing system, with new run-time mechanisms for atomicity across different consistency levels.

### 2.3 Mixing Consistency Breaks Strong Consistency

There is a reason that existing systems choose a single consistency level for each transaction. Causal consistency and linearizability offer well-defined consistency guarantees, but trying to mix these levels in the same application can break the guarantees that both levels claim to offer, even if all the issues in the previous section were solved. Some transactions simply are not safe to run under mixed consistency. To see why, consider the following example.

Suppose we run a contest to advertise Message Groups. Users are divided into two teams; team A sends messages to mailbox `a`, and team B sends messages to mailbox `b`. The first team to send 1,000,000 messages is declared the winner.

To implement this contest, we extend the existing transaction for delivering mail with a few lines of code shown in Figure 2. After running the contest, we may be surprised to discover that the code has not declared a unique winner; both teams A and B are simultaneously declared the winner!

This code has a fundamental problem. To avoid slowing down the core functionality of message delivery, the guard condition uses data (the inbox sizes) stored with only causal consistency. Since the guard is evaluated with causal consistency, nothing guarantees that the function `declare_winner()` is invoked only once. But the function `declare_winner()` manipulates only data with linearizable consistency; it is not designed to deal with the potential for multiple re-executions. During a partial network partition, every single message receipt to either team could cause the winner to switch, as the causal replica receiving messages for `a` may not be able to propagate events to the replica receiving messages for `b` (and vice-versa). This causes each team to believe their inbox alone has reached the target size.

The essence of this mistake is that more-consistent data (the declared winner) is influenced by less-consistent data (the inbox size). This inappropriate influence means the developers of `declare_winner()` would have to add complex code to ensure its assumptions hold under weak consistency guarantees, even though `declare_winner()` does not access weakly consistent data itself.

The issue of weakly consistent data influencing strongly consistent computations is fundamental to the semantics of consistency. Even within a linearizable transaction, the influence of weakly consistent data on program control flow can effectively weaken the isolation level of the entire

$$
\begin{aligned}
x &\in \textbf{Var} \qquad f \in \textbf{Operation} \\
\oplus &\in \textbf{Binop} \qquad \ominus \in \textbf{Unop} \\
(\textit{Location})\ m &::= x \mid *e \mid e.x \mid e \texttt{->} x \\
(\textit{Expr})\ e &::= m \mid e_1 \oplus e_2 \mid \ominus e \mid e_0.f(e_1, \ldots, e_n) \\
(\textit{Stmt})\ s &::= \texttt{var}\ x = e \mid m = e \mid \texttt{return}\ e \\
&\quad \mid \texttt{while (}e\texttt{)}\ s \mid \texttt{if (}e\texttt{)}\ s_1\ \texttt{else}\ s_2 \mid \texttt{\{}s_1, \ldots, s_n\texttt{\}}
\end{aligned}
$$

Fig. 3. MixT surface syntax. Certain built-in operations are omitted for clarity of exposition.

transaction. MixT uses information flow analysis to flag such influences at compile time, disallowing this example code. As discussed in Section 7, MixT also allows intentional weakening of this restriction.

## 3 MIXT TRANSACTION LANGUAGE

We solve the problems introduced in the previous section with MixT, a new domain-specific language (DSL). To support a variety of underlying stores in a uniform way, including key–value stores, databases, and file systems, MixT offers a high-level embedded transaction language that is straightforward to adapt to new stores.

### 3.1 MixT Language Syntax

Figure 3 gives the surface syntax of the MixT language. Because MixT is embedded in C++, its syntax and semantics, though different from those of its host language, are designed to be unsurprising to C++ programmers.

MixT is relatively expressive; for example, it has control structures like conditionals and loops. Despite supporting real control structures, MixT transactions are fully atomic when the underlying stores support atomic transactions. In particular, all transaction effects become visible at once, and transactions operate against stable snapshots at each store. Like C++, the MixT language has mutable locations, which can be either local variables or fields of objects.

Though they are not shown explicitly in Figure 3, *handles* are a key abstraction of MixT. Handles behave like pointers to remotely stored persistent data; they can be dereferenced to access the underlying data (with the operators `*` and `->`), and they can be aliased by assignment.

Handles also support the invocation of operations on data. Given a handle $e_0$ to a receiver object, the expression $e_0.f(e_1, \ldots, e_n)$ invokes a *custom operation* named $f$, provided by the underlying store of the receiver.[1] Exactly which operations are supported depends on the store. For example, many stores provide specialized operations for manipulating sets, but even SQL queries can be exposed as operations.

```
1 class user {
2   Handle<set<string>, causal, supports<insert>> inbox;
3 };

5 class group {
6   RemoteList<Handle<user, causal>, linearizable> users;
7   Handle<Log, eventual, supports<append>> log;
8   mixt_method(add_post) (post) mixt_captures(users,log) (
9     var iterator = users,
10    while (iterator.isValid()) {
11      log.append(iterator->v.inbox.insert(post)),
12      iterator = iterator->next
13    }
14  )
15 };
```

Fig. 4. MixT message delivery implementation (§4.2). MixT code (lines 8–14) is colored green; C++ code is blue.

### 3.2 A MixT Example Program

As an example of MixT code used within a larger program, the message delivery transaction of Section 2.2 is shown in Figure 4. To distinguish MixT code from surrounding C++ code, MixT code is colored green, whereas C++ code is blue.

Most of this code should look familiar to a C++ programmer; outside the transaction, it merely defines classes that contain library types, such as the MixT library type `RemoteList`, as fields.

At the heart of MixT are transaction blocks, signified by the `mixt_method` declaration. For example, at lines 8–14 is the now-familiar transaction for message delivery, expressed as a method `add_post()` of the C++ class `group`. This method can be invoked from any context without the need to explicitly start a transaction; its parameter `post` is automatically inferred to be a string.

In this transaction, the expressions `iterator`, `inbox`, `users`, and `log` are all handles for state on remote stores. The type `Handle<T, L, ...>` is the C++ representation of a MixT handle for data of type `T`, stored at consistency `L`. An object of this class acts as an opaque representation of a persistent resource. Any supported custom operations appear in the third and following argument positions. For example, `inbox` (line 2) is a set of strings, stored at causal consistency, with a custom operation `insert` for adding new items to the set. It is the job of the causal store to ensure that `insert` operations from different clients are merged with causal consistency.

MixT offers some useful data structures as library types. For example, the type `RemoteList`, used at line 6, is a persistent linked list that stores its spine at a specified consistency level (here, linearizable).

### 3.3 MixT API

As much as possible, MixT operates as a shim above existing stores, reusing their existing mechanisms for replication and data consistency. It is straightforward to add support for a new store,

[1]The store may specify whether its parameters should be treated as opaque handles, arbitrary values, or dereferenced handles to other objects on this store. The arguments $e_1, ..., e_n$ are passed as values by default, except when the store requests otherwise, at which point they will be dereferenced (resulting in a run-time error if these arguments refer to handles on the wrong store).

```
class Handle<Type, Label, Operations...> {
    Type get(TransactionContext);
    void put(TransactionContext, Type);
    bool isValid(TransactionContext);
    Type clone(TransactionContext);
};
class DataStore<Label> {
    TransactionContext beginTransaction();
};
class TransactionContext {
    bool commit();
    DataStore store();
};
```

Fig. 5. Handle and DataStore interfaces.

```
class CausalStore : public DataStore<causal> {
  template<typename T>
  class CausalObj : CausalHandle<T> {...};
  template<typename T>
  mixt_operation(insert) (CausalObj<set<T>>&, T&) {...}
  ...
};
```

Fig. 6. Implementing a causal store in a host C++ program.

as long as it offers the necessary functionality; one simply implements three interfaces, `Handle`, `DataStore`, and `TransactionContext`, shown in Figure 5.

The `Handle` interface consists of a simple get/put/check API for accessing underlying data, a set of routines for supporting marshaling, and a set of routines for accessing and using the store from which the `Handle` originated. Much of this functionality can be automatically generated by the MixT libraries at compile time; Figure 5 only includes routines the programmer must implement.

A `DataStore` serves as an entry point to the underlying storage system; it is always associated with a specific consistency label (level) and a specific implementation of `Handle`. The only requirement from the `DataStore` API is the method `beginTransaction()`, which must create a new transaction represented by a `TransactionContext` object. The `TransactionContext` can be used to commit or abort the transaction interactively, and can be extended to supply store-specific transaction interactions options. A `DataStore` may also implement any number of custom operations, ranging in complexity from creating new remote objects to processing SQL statements.

Figure 6 illustrates how a causal store with a custom operation `insert` can be implemented. Custom operations are declared within classes implementing `DataStore` by using the `mixt_operation` keyword. In this example, the operation `insert` is declared to take a remote `set` and a local `T`, matching the types on which it was invoked in `add_post` (Figure 4). Unlike `mixt_method`, `mixt_operation` does *not* declare a C++ method, and can only be called from within a MixT transaction. Within a transaction, operations dynamically dispatch to the appropriate `Handle` and `DataStore`; to facilitate this dispatch, every `Handle`'s type also includes a static list of operations which its implementation supports.

MixT custom operations provide a method-like syntax for invoking operations directly on handles to remote data, as with `insert` in Figure 4. It would be a mistake, however, to imagine that they are limited only to "method-like" invocations directly on remote data; they are flexible enough to expose arbitrary database functionality directly to a MixT transaction. For example, one could create a `Handle<DB>`, with matching `mixt_operation`s for interfacing directly with the underlying database's raw API. If the database exposed more stateful functionality, such as locks, a `Handle<DBLock>` could be used to manage each individual lock.

## 4 MIXED-CONSISTENCY TRANSACTIONS

Section 2 shows that even seemingly trivial code can require the implementer to reason very carefully about the interactions between different consistency levels in the presence of possible transaction aborts. The complexity of this reasoning can easily become overwhelming. MixT tames this complexity by providing semantics for mixed-consistency transactions (§4.1). MixT's transaction support can provide atomic execution for Section 2.1's message delivery transaction (§4.5), and its type system will detect the fundamental errors of Section 2.3's contest (§4.3). A more detailed look at MixT's transactions comes in Section 5.

### 4.1 Defining Mixed Consistency

We now address a fundamental question: what are the desired semantics of mixed consistency? We choose the standard approach used for shared-memory consistency [27], in which a consistency model is characterized as a trace property: that is, as the (possibly infinite) set of execution traces that do not violate the consistency model's guarantees. In principle, we can then verify whether a program execution satisfies a given model by checking whether its trace is in the set.

In mixed-consistency transactions, objects labeled with some consistency model should enjoy at least the guarantees of that model. For example, in a system with both eventual consistency and linearizability, traces involving any subset of objects should adhere at least to eventual consistency, and traces involving only its linearizable objects should respect linearizability. Put another way, an observer who accesses only linearizable objects should be unable to determine that there are any eventually consistent operations in the system.

The strength of consistency models can be characterized in terms of the possible behaviors of programs. The behaviors of the programs form a set of admitted traces $T$. The meaning of a consistency level $\ell$ is given by its consistency model, a set of traces $T_\ell \subseteq T$. A model $T_\ell$ is stronger than a model $T_{\ell'}$ when $T_\ell \subseteq T_{\ell'}$; $T_\ell$ provides more guarantees than $T_{\ell'}$. All consistency models must include the empty trace. We assume there is a lattice of consistency levels $\mathcal{L}$ ordered by strength. If a consistency level $\ell$ is stronger than or equal to another, $\ell'$, we write $\ell \sqsubseteq \ell'$. Consistency models are ordered by inclusion consistently with the ordering on $\mathcal{L}$: $\ell \sqsubseteq \ell' \iff T_\ell \subseteq T_{\ell'}$, $T_{\ell \sqcup \ell'} = T_\ell \cup T_{\ell'}$, and $T_{\ell \sqcap \ell'} = T_\ell \cap T_{\ell'}$.

Each trace $t \in T$ is a sequence of events $e$. An event $e$ is a 5-tuple $(a, o, \ell, v, S)$ containing $a$, the action corresponding to this event; $o$, the exact memory location or object referenced by this event; $\ell$, the consistency level of the store for this event's location; $v$, a tuple of any values processed by this event; and $S$, the client session in which this event occurred. Given such an event, we define $\mathit{consistency}((a, o, \ell, v, S)) = \ell$. For example, the program `x = 4; x = x + 1`, wherein `x` resides on a store with consistency $\ell$, admits the trace "(write, $x$, $\ell$,(4), $S$); (read, $x$, $\ell$, (4), $S$); (write, $x$, $\ell$, (5), $S$)" when executed in session $S$.

Given a trace $t$, the events relevant to a given consistency level $\ell$ are those whose consistency level is at least as strong. We write $t \downharpoonright \ell$ to denote the trace containing such events:

*Definition 4.1 (Trace projection).*

$$t \downarrow \ell = [e \mid e \in t \wedge \mathit{consistency}(e) \sqsubseteq \ell]$$

*Definition 4.2 (Mixed consistency).* A trace $t$ exhibits *mixed consistency* if it satisfies *every* consistency model $T_\ell$ when projected onto that consistency level:

$$\forall \ell, t \downarrow \ell \in T_\ell$$

This definition is sensible even when working with incomparable consistency models; because consistency models form a lattice [54], there is always some minimum consistency model onto which all events can be projected.

Definition 4.2 can also be adapted to transaction isolation levels [4] by considering traces containing explicit events that begin and end transactions. A full formalization is found in Section 6.

### 4.2 Noninterference for Mixed Consistency

In Section 4.1, we proposed a definition for *mixed consistency* based on the approach used for shared-memory consistency. But this approach can hide influence: common consistency models, expressed in terms of reads and writes to shared registers, are not strong enough to capture *why* each read or write occurs. To capture this influence directly, we look to *noninterference*, a semantic property common in the security and privacy literature. Noninterference describes programs as secure if, when given a policy lattice of security labels, program behavior at one point in the policy lattice cannot influence behavior at levels that are lower in the lattice or incomparable [22, 50]. In particular, when using noninterference to enforce privacy or confidentiality, two runs of a program that differ only in secret inputs should have identical publicly observable behavior. Noninterference is the correcntess condition normally associated with information flow (Section 4.3).

We start by taking this traditional approach, replacing secret with "weakly consistent" and public with "strongly consistent;" in other words, we determine if any "weakly consistent" data can influence any "strongly consistent" data by comparing the possible runs of transactions. We cannot simply compare pairs of runs, however, because systems built using MixT are inherently concurrent and nondeterministic; two runs may differ simply as a result of acceptable nondeterminism. We instead consider *sets* of possible runs generated by keeping the deterministic program inputs fixed, but varying the nondeterministic choices made by the program. We say that weakly consistent data has an improper influence in this program if varying weakly consistent data introduces new strongly consistent data into the set. Put another way, varying weakly consistent data should only affect strongly consistent values in ways already permitted by the inherent nondeterminism of the system. This *possibilistic* notion of information flow is called generalized noninterference [43].

Possibilistic security has been shown to be problematic in its original setting of confidentiality, because information can be leaked via refinement [55, 63]. In the context of consistency and other integrity-like properties, it does not seem to be a major concern [40].

### 4.3 Consistency as Information Flow

To enforce generalized noninterference, we treat consistency as a form of information-flow integrity [5] and use an information-flow type system [50] to outlaw bad programs. Previous work [55] has shown that generalized noninterference is soundly enforceable using this style of security type system. In such a type system, values are associated with a label drawn from a lattice, which in this case is a lattice of consistency levels. The strongest possible consistency is the lowest point in the lattice, denoted $\bot$, and the weakest consistency is $\top$. To enforce consistency, information should not be influenced by other information whose consistency is not at least as strong. Therefore, as in other work on information flow, legal information flow is upward in the lattice.

In the case of the buggy contest in Section 2.3, the transaction creates a banned information flow from the inbox size (weak) to the `declare_winner()` operation (strong). In information-flow terms, this is an implicit flow [50]. The type system of MixT (Section 5.2) statically catches invalid flows, whether implicit or explicit, and rejects unsafe transactions.

### 4.4 Transaction Splitting

We now turn to the difficult task of implementing noninterfering transactions against multiple backing databases. Consider again the message delivery code in Figure 4. This code is noninterferent and is therefore safe in principle, but because it involves three different consistency levels, it is nonetheless quite difficult to implement, as discussed in Section 2.2.

MixT implements mixed-consistency transactions like this one by automatically splitting their code into a single sub-transaction per involved store. A key insight is that safe splitting is always possible because information flow restrictions prevent weakly consistent data from affecting strongly consistent data either directly or indirectly within a transaction. Hence, transactions can be split so that their stronger-consistency parts are executed earlier. This allows each sub-transaction to be safely re-executed in the case of a transaction abort, avoiding the pitfalls inherent to partitioning data across systems outlined in Section 2.2. This splitting does not automatically preserve atomicity, the subject of Section 4.5.

In general, a *split transaction* consists of a sequence of syntactically separate transaction phases. For each consistency level in the transaction, there is a single phase for all operations with that consistency level. MixT determines which data are communicated between phases, preserving only the information necessary to execute subsequent phases.

For example, the message delivery transaction is split into linearizable, causal, and eventual phases (in order of decreasing strength of consistency guarantees), corresponding to the consistency levels used by the transaction.

The most challenging aspect of transaction splitting is the treatment of loops, which makes splitting quite different than in previous work on automatic transaction splitting [11, 64]. Like all expressions, each loop's condition must be evaluated within a single phase, but the body of the loop might contain statements that execute in different phases. A loop spanning multiple consistency levels, such as in the message delivery transaction, must therefore be re-executed for each consistency level.

The information-flow type system ensures that all statements that affect the loop's condition occur at the first and strongest phase in which the loop appears. In this first phase, MixT explicitly records the results of each conditional test for the loop, replaying them during the loop execution in subsequent phases. For more detail about this process and a worked example, see Section 5.3.

### 4.5 Whole-Transaction Atomicity

Static transaction splitting produces a single sub-transaction per underlying store, allowing us to inherit the guarantees of isolation and atomicity provided for all operations on that store. Splitting does not, however, guarantee atomicity for the entire transaction, since commits to stronger stores happen before commits to weaker ones. To ensure atomicity, MixT programs must be prevented from observing the effects of partially committed transactions. When atomicity is guaranteed by at most one of the stores to which a transaction writes, no extra machinery is needed. However, for the rare transaction that writes to multiple atomic stores, we introduce *witnesses*, which lock affected objects.

During each phase's execution, MixT creates a special *write witness* object for each mutation, indicating that a lock has been acquired on the object being mutated. At the end of each phase, MixT creates a single *commit witness*, a special object which indicates that all locks acquired during

$$
\begin{aligned}
(\mathit{Location})\ m &::= x \mid m.x \\
(\mathit{Expr})\ e &::= m \mid x_1 \oplus x_2 \mid \ominus x \mid x_0.f(x_1, \ldots, x_n) \\
(\mathit{Stmt})\ s &::= \texttt{var}\ x\ =\ e\ \texttt{in}\ s \mid \texttt{remote}\ x\ =\ e\ \texttt{in}\ s \\
&\quad \mid m\ =\ x \mid \texttt{return}\ x \mid \texttt{while}\ (x)\ s \\
&\quad \mid \texttt{if}\ (x)\ s_1\ \texttt{else}\ s_2 \mid \{s_1, \ldots, s_n\}
\end{aligned}
$$

Fig. 7. MixT flattened syntax.

this transaction have been released. Only one witness is produced per transaction, but a copy of it is sent to every store on which writes were performed. If a MixT transaction encounters a write witness, it must suspend execution until it encounters the corresponding commit witness.

The witness mechanism ties together phases of split transactions across mutually unaware systems. By creating an explicit object during each transaction and blocking future progress until it has appeared, we guarantee atomicity; the full transaction will be visible to all future transactions.

The witness mechanism should impose relatively little overhead because its use should be rare. Further, several optimizations (Section 8.1) can reduce its overhead. The performance evaluation (Section 9) shows that a complex MixT program can achieve reasonable throughput even when witnesses are used. We revisit witnesses in more detail in Section 5.4; formal arguments regarding the correctness of witnesses can be found in Section 6.

## 5 FORMALIZING THE MIXT LANGUAGE

### 5.1 Desugared Language

To facilitate transaction splitting, MixT's surface syntax is translated to a "flattened" language whose syntax appears in Figure 7. Where the surface and flattened languages coincide, they share the same semantics. There are a few notable changes from the surface language. All expressions are flattened by the compiler using standard techniques [51]. The pointer-like syntax $*e$ and $e\texttt{->}x$ is replaced by the ability to declare `remote` variables bound to handles. Semantically, `remote` variables directly correspond to the referenced location on an underlying store. Updates to these variables are reflected at the store, and uses of these variables query the store directly for their value. Unlike in the surface language, both `var` and `remote` introduce explicit scopes for their bindings.

### 5.2 Statically Checking Consistency Labels

Consistency is enforced in MixT by statically checking information flow using a largely standard type system for static information flow [50].

Figure 8 gives selected consistency typing rules for the language. Ordinary rules for typing judgments $\Gamma \vdash e : \tau$ are not presented because they directly use the C++ type system; the presented rules are only for *consistency judgments* $\Delta \mid \Gamma \mid pc \vdash e : \ell$. Environments $\Delta$ and $\Gamma$ keep track of the labels and types of variables, respectively, with local and remote variables distinguished lexically by subscripts $V$ and $R$. The label $pc$ (for *program counter*) bounds the consistency of control flow.

The rules assign each statement and expression a consistency label $\ell$ that reflects the weakest consistency of any information used to compute it. The label on statements, used during transaction splitting, is derived directly from subexpressions and is unaffected by substatements. During static checking, consistency originates from the consistency labels on handles, which derive from their stores. Variables captured from the environment outside of the transaction are labeled with the strongest ($\bot$) consistency; all other labels are automatically inferred from the transaction code.

$$\frac{\Delta \mid \Gamma \mid pc \vdash e : \ell_1 \qquad \Gamma \vdash e : \tau \qquad \Delta, x_V : \ell \mid \Gamma, x_V : \tau \mid pc \vdash s : \ell_2 \qquad \ell_1 \sqsubseteq \ell \qquad x \notin \Gamma \qquad x \notin \Delta}{\Delta \mid \Gamma \mid pc \vdash \texttt{var}\ x\ \texttt{=}\ e\ \texttt{in}\ s : \ell}$$

$$\frac{\Gamma \vdash e : \texttt{Handle}\langle \tau, \ell_1 \rangle \qquad \Delta \mid \Gamma \mid pc \vdash e : \ell_2 \qquad \Delta, x_R : \ell \mid \Gamma, x_R : \tau \mid pc \vdash s : \ell' \qquad \ell_1 \sqcup \ell_2 \sqsubseteq \ell \qquad x \notin \Gamma \qquad x \notin \Delta}{\Delta \mid \Gamma \mid pc \vdash \texttt{remote}\ x\ \texttt{=}\ e\ \texttt{in}\ s : \ell}$$

$$\frac{\Delta, x\_ : \ell \mid \Gamma \mid pc \vdash e : \ell' \qquad \ell' \sqsubseteq \ell}{\Delta, x\_ : \ell \mid \Gamma \mid pc \vdash x\ \texttt{=}\ e : \ell} \qquad \frac{pc \sqsubseteq \ell}{\Delta, x_R : \ell \mid \Gamma \mid pc \vdash x : \ell}\ \textsc{remote-read} \qquad \Delta, x_V : \ell \mid \Gamma \mid pc \vdash x : \ell$$

$$\frac{\Delta \mid \Gamma \mid pc \vdash e : \ell' \qquad \ell' \sqsubseteq \ell}{\Delta \mid \Gamma \mid pc \vdash e : \ell} \qquad \frac{\Delta \mid \Gamma \mid pc \vdash e : \ell \qquad \Delta \mid \Gamma \mid pc \sqcup \ell \vdash s : \ell'}{\Delta \mid \Gamma \mid pc \vdash \texttt{while}\ (e)\ s : \ell}$$

$$\frac{\Delta \mid \Gamma \mid pc \vdash e : \ell \qquad \Delta \mid \Gamma \mid pc \sqcup \ell \vdash s_1 : \ell' \qquad \Delta \mid \Gamma \mid pc \sqcup \ell \vdash s_2 : \ell'}{\Delta \mid \Gamma \mid pc \vdash \texttt{if}\ (e)\ s_1\ \texttt{else}\ s_2 : \ell} \qquad \frac{pc \sqsubseteq \bot}{\Delta \mid \Gamma \mid pc \vdash \texttt{return}\ (e) : \top}$$

Fig. 8. Selected consistency typing rules for MixT. The labeled REMOTE-READ rule is unusual.

One non-standard aspect of the rules is that all accesses to `remote`-bound variables are treated as effectful, requiring the same *pc* consistency to read a `remote` location as to write it (REMOTE-READ). This restriction is imposed for correct transaction splitting, to enforce the necessary condition that all remote operations at a single consistency level execute before any remote operations at a weaker level.

Omitted from Figure 8 is the rule governing endorsement, discussed in Section 7.

### 5.3 Transaction Splitting

Figure 9 gives selected rules for splitting transactions into phases based on consistency labels. The translation $[\![\cdot]\!]_\ell$ generates the code for the transaction phase at consistency level $\ell$.

Recall that each statement in the flattened language is associated with exactly one consistency level. Intuitively, transaction splitting preserves a statement in phase $\ell$ when the statement's label matches $\ell$ and otherwise omits it from the phase. However, statements associated with a nested scope, such as `while` and `var`, may execute their contained statements in a different phase.

Once some variable $x$ is introduced in a phase, it may be used—but not assigned—in any later phase. During the phase in which $x$ is introduced, every binding and assignment to $x$ is stored in an implicit iterator. During subsequent phases, this iterator is used to replay $x$'s previous values. In these subsequent phases, uses of $x$ are replaced with `peek(`$x$`)` expressions, which return the current value of $x$'s implicit iterator. Mutations to $x$ are replaced with `advance(`$x$`)`, which advances $x$'s implicit iterator. Remote-bound variables require the additional `advance binding(`$x$`)` construct. Recall that the `remote` $x$ `=` $e$ construct binds $x$ as an alias to the remote state described by $e$. If this binding appears in a loop, then the value of $e$ may shift, causing $x$ to be bound to a series of remote locations. The `advance binding(`$x$`)` statement cycles through these, while the `advance(`$x$`)` statement cycles through assignments to the remote object itself. These and other syntactic extensions required by transaction splitting are shown in Figure 9.

For example, the split transaction generated from the message delivery transaction contains three non-local phases, one for each distinct consistency level used, as shown in Figure 10 (for

$$
\begin{aligned}
(\textit{Expr})\ e &::= \ldots \mid \texttt{rand()} \mid \texttt{peek}(x) \\
(\textit{Stmt})\ s &::= \ldots \mid \texttt{release\_all}(n) \\
&\quad \mid \texttt{acquire}(x, n, \ell_1, \ldots, \ell_n) \\
&\quad \mid \texttt{advance}(x) \mid \texttt{advance remote}(x) \\
(\textit{Phase})\ p &::= s_\ell \\
(\textit{Transaction})\ t &::= \mathcal{T}\{p_1; p_2; \ldots; p_n\}
\end{aligned}
$$

$$
[\![\texttt{remote}\ x = e\ \texttt{in}\ s : \ell]\!]_\ell \triangleq \texttt{remote}\ x = e\ \texttt{in}\ [\![s]\!]_\ell \qquad \frac{\ell \not\sqsubseteq \ell'}{[\![\texttt{remote}\ x = e\ \texttt{in}\ s : \ell]\!]_{\ell'} \triangleq [\![s]\!]_{\ell'}}
$$

$$
\frac{\ell \sqsubseteq \ell'}{[\![\texttt{remote}\ x = e\ \texttt{in}\ s : \ell]\!]_{\ell'} \triangleq \{\texttt{advance binding}(x), [\![s]\!]_{\ell'}\}}
$$

$$
[\![\texttt{while}\ (e)\ \mathit{stmt} : \ell]\!]_\ell \triangleq \texttt{while}\ (e)\ [\![\mathit{stmt}]\!]_\ell \qquad \frac{\ell \not\sqsubseteq \ell'}{[\![\texttt{while}\ (e)\ \mathit{stmt} : \ell]\!]_{\ell'} \triangleq \{\}}
$$

$$
\frac{\ell \neq \ell' \qquad \ell \sqsubseteq \ell'}{[\![x = e : \ell]\!]_{\ell'} \triangleq \texttt{advance}(x)} \qquad \frac{\ell \neq \ell' \qquad \ell \sqsubseteq \ell'}{[\![x : \ell]\!]_{\ell'} \triangleq \texttt{peek}(x)} \qquad [\![x : \ell]\!]_\ell \triangleq x
$$

$$
\frac{L = \{\ell_1, \ell_2, \ldots, \ell_n\}}{\mathcal{S}[\![\mathit{stmt}]\!]_L \triangleq \mathcal{T}\{[\![\mathit{stmt}]\!]_{\ell_1}; [\![\mathit{stmt}]\!]_{\ell_2}; \ldots; [\![\mathit{stmt}]\!]_{\ell_n}\}}
$$

Fig. 9. Selected transaction splitting rules.

clarity, the code is presented in the surface syntax). Updates to the freshly generated variables `loopindex`, `temporary1`, and `temporary0` are logged; the expression `peek(`$x$`)` accesses an iterator over the previous values of $x$, and the expression `advance(`$x$`)` advances this iterator. Neither of the original variables `iterator` and `users` is necessary for any subsequent phase, so they are discarded upon successful commit of the first phase.

### 5.4 Enforcing Atomicity in Split Transactions

After transaction splitting, the MixT compiler augments the split transaction with code to create witnesses. First, the transaction generates a new variable `wit` (as shown in the GENERATE rule of Figure 11) . We use this as the name of our commit witness. To avoid collisions, this name is selected randomly from a reserved 63-bit namespace. Next, it inserts the statement `acquire(`$x$`,wit,`$\mathit{phases}$`)` before all mutative operations (ACQUIRE rules). This statement creates a write witness, writing it alongside $x$. At the end of the phase, the compiler appends `release_all(wit)` (RELEASE rule) , which writes the commit witness itself. Witnesses are only required for transactions which perform writes to remote storage in multiple phases (captured in the precondition to the GENERATE rule) . Witnesses are optional; if a user does not require full transaction atomicity, they can opt out of witness generation.

As an example, we present witness generation for the familiar message delivery transaction in Figure 1. A new first phase contains witness generation; in all phases, each mutative operation is immediately followed by a call to `acquire`. At the end of each phase, MixT copies the commit witness to the store via `release_all`.

```
var iterator = users, // Phase: strict    serializability
var loopindex = iterator.isValid(),
while (loopindex) {
  loopindex = iterator.isValid(),
  var temporary0 = iterator->v,
  iterator = iterator->next
}
```

---

```
advance(loopindex), // Phase: causal   consistency
while (loopindex) {
  advance(loopindex),
  advance(temporary0),
  var temporary1 = peek(temporary0).inbox.insert(post)
}
```

---

```
advance(loopindex), // Phase: eventual   consistency
while (loopindex) {
  advance(loopindex),
  advance(temporary1),
  logger.log(peek(temporary1))
}
```

Fig. 10. The message delivery transaction after splitting, lifted back to the surface syntax.

$$\text{GENERATE}\quad \frac{p_i, p_j \in \mathcal{P}.p_i \neq p_j \wedge \mathit{writes}(p_i) \wedge \mathit{writes}(p_j) \qquad \mathcal{P} = [p_1, \ldots, p_n]}{[\![\,[p_1, \ldots, p_n]\,]\!] \triangleq [\texttt{var wit = gensym()}_\perp, [\![p_1]\!]_2, \ldots, [\![p_n]\!]_2]}$$

$$\text{RELEASE}\quad [\![\mathit{phase}_n]\!]_2 \triangleq [\![\mathit{phase}_n]\!]_3; \texttt{release\_all(wit)}$$

$$\text{ACQUIRE1}\quad \frac{\text{x is remote-bound}}{[\![\texttt{x = e}]\!]_3 \triangleq \texttt{x=e; acquire(x,wit,}\mathcal{P}\texttt{)}}$$

$$\text{ACQUIRE2}\quad [\![\texttt{x.f(...)}]\!]_3 \triangleq \texttt{x.f(...); acquire(x,wit,}\mathcal{P}\texttt{)}$$

Fig. 11. Selected rules for witness modification. $\mathcal{P}$ is the list of transaction phases generated during splitting of a MixT transaction, and *writes* is a boolean function indicating whether a write is performed in a phase. Semicolon separates instructions.

The purpose of witness transformation is simple: each call to `acquire(`$x$`, wit, ...)` acquires a logical lock on $x$, which is released by the corresponding call to `release_all(wit)`. Any transaction which observes the "lock acquire" event (`acquire`) must now wait for the corresponding "lock release" event (`release_all`) before proceeding at each participating phase.

Witnesses are implemented as simple objects stored directly on remote stores. Write witnesses contain the locations of all corresponding commit witnesses and a list of all consistency levels involved in the transaction, while commit witnesses are empty objects.

Additional run-time checks beyond those inserted by Figure 11 are required to check for witnesses. At run time, whenever MixT attempts to read a `remote`-bound value, it also reads that value's

```
var wit = rand()
```

---

```
var iterator = users, // Phase: linearizable
var loopindex = iterator.isValid(),
while (loopindex) {
  loopindex = iterator.isValid(),
  var temporary0 = iterator->v,
  acquire(temporary0,wit,linearizable,causal,eventual),
  iterator = iterator->next
}
release_all(wit)
```

---

```
advance(loopindex), //Phase: causal consistency
while (loopindex) {
  advance(loopindex),
  advance(temporary0),
  var temporary1 = peek(temporary0).inbox.insert(post),
  acquire(temporary1,wit,linearizable,causal,eventual)
}
release_all(wit)
```

---

```
advance(loopindex), //Phase: eventual consistency
while (loopindex) {
  advance(loopindex),
  advance(temporary1),
  logger.log(peek(temporary1))
}
release_all(wit)
```

Fig. 12. The message delivery transaction after witness insertion, lifted back to the surface syntax

```
class DataStore<Label> {
    ...
    bool exists(Name name);
    Handle<...> existingObject(Name name);
    Handle<...> newObject(Name name);
};
```

Fig. 13. Enhanced DataStore interface for witnesses.

potential witness location, checking for a write witness. If it finds one, it adds the discovered commit witness's location to a *witness worklist* for each phase listed in the write witness.

Before MixT begins executing a phase $p$, it first iterates through $p$'s witness worklist. For each commit witness $w$ in the worklist, MixT polls $p$'s store until $w$ becomes available, and then removes $w$ from the worklist. As described in Section 8.2, this polling loop occurs on the remote store itself. Once the commit witness has appeared on all replicas, it may be safely removed from the store; the specifics of this process are found in Section 8.1.

These additional run-time checks prevent a transaction from observing its own causal past; if a certain transaction phase has witnessed a past transaction, then we must ensure that all future transaction phases can also witness that transaction.

To support witnesses, we extend the requirements on underlying storage to include a key–value API through which witnesses can be named (Figure 13). These APIs take the form of type constraints; the return and parameter types must support certain operations, but do not need to be a precise MixT-specified type.

Note that using witnesses does not cause the resulting transaction to become fully serializable; if a weak consistency level allows stale reads, then stale reads can still occur during the weak phase of a mixed-consistency transaction. A mixed transaction might fail to read values committed by a previous fully weak transaction. Even with witnesses, the code in Figure 2 remains incorrect. Also, atomicity is only guaranteed when the underlying store provides it; consistency levels which do not have a useful notion of atomicity—for example, eventual consistency—therefore do not employ write or commit witnesses.

## 6 CORRECTNESS

### 6.1 Mixed isolation

To argue correctness of mixed-consistency transactions, we must first establish a notion of *mixed isolation* levels. As with consistency models, we represent isolation levels as trace properties. To have a common framework in which to discuss both consistency and isolation, we associate each consistency level $\ell$ with an isolation level $\mathcal{I}_\ell$. We again characterize the behavior of the program as a set of admitted traces $T$, where each trace $t \in T$ is a sequence of events $e$. An event $e$ is now a 6-tuple $(a, o, \ell, \mathcal{T}, v, S)$; members of this tuple are as defined in Section 4.1 with the exception of $\mathcal{T}$, a unique token corresponding to the transaction to which this event belongs. We lift $\downarrow$ to this new setting as $\downarrow_{\mathcal{I}}$. We define $\mathcal{T}_e = \pi_4(e)$ as the transaction of an event. Our traces now include explicit events for transaction begin ($begin_{\mathcal{T}}$), abort ($abort_{\mathcal{T}}$), and commit ($commit_{\mathcal{T}}$), along with corresponding actions *begin*, *abort*, and *commit*. Similarly to consistency, the meaning of each isolation level $\mathcal{I}_\ell$ is given by its isolation model $I_\ell$, an (infinite) set of traces containing all executions which satisfy the guarantees of the isolation level. We use $<$ as the order for events within traces.

We define a forgetful function $F(t) = \{(a, o, \ell, v, S) \mid (a, o, \ell, \mathcal{T}, v, S) \in t \wedge a \notin \{end, commit, abort\}\}$ which drops transaction events from a trace, converting it back to Section 4.1's notion of application traces. We say that an isolation model $I_\ell$ is well-formed if $\{F(t) \mid t \in I_\ell\} \subseteq \ell$; that is, normal events in the isolation model are also in its associated consistency model. We now define *mixed isolation* as an analog of *mixed consistency*:

*Definition 6.1 (Mixed isolation of trace t).*

$$\forall \ell, t \downarrow_{\mathcal{I}} \ell \in I_\ell$$

We now proceed to argue that our mechanisms of transaction splitting and witnesses are sufficient to guarantee both mixed consistency and atomicity for mixed isolation.

### 6.2 Transaction splitting ensures mixed consistency and noninterference

We first argue that the trace generated by any sequence of transactions which do not utilize the `return` or `endorse` constructs are guaranteed to preserve mixed-consistency. This argument is naturally dependent on the nature of the consistency model in question. If the consistency model, as is traditional, includes only information about independent reads and writes made to individual memory locations then it is easy to see that MixT satisfies mixed consistency. Because MixT associates each data object or memory location with a single consistency model, all reads of some value from a memory location at some consistency level $\ell$ must be paired with a write to that location at level $\ell$. Thus, for all reads in a trace, the projection operator ($\downarrow$) also preserves all

matching writes in that trace. In this sort of consistency model, the projection operator simply selects sets of memory locations.

We next turn to enforcing noninterference (again, for transactions without `endorse` or `return`). This follows directly from our splitting algorithm and type system. In MixT, labels are derived only from remote actions; during transaction splitting, these remote actions are segregated into phases. For any pair of labels $\ell_w \sqsubset \ell_q$, all $\ell_q$-labelled operations occur before $\ell_w$'s phase, and thus $\ell_w$ actions cannot interfere with $\ell_q$ actions. For pairs of incomperable labels $\ell_l$ and $\ell_r$, our compiler chooses a deterministic order in which to execute their phases, and our runtime executes both phases with the environment of their last ancestor. Thus even in this case, our phases cannot interfere. The remote actions themselves are expected to be carried out on distinct, mutually-unaware backing storage systems; these systems are strongly isolated from each other, which prohibits any additional influence channels.

We make no argument that the trace of an entire program will adhere to our properties; as MixT is embedded within the context of a larger C++ program, it will always be possible for programmers to take the value returned by one transaction, ignore its consistency, and use it inappropriately in a subsequent transaction.

### 6.3 Write witnesses ensure atomicity

We now address the claim that our split transactions preserve atomicity, given underlying stores which themselves preserve atomicity. As the strongest purely atomicity-based property granted by a standard isolation level is read committed [1], we limit ourselves to demonstrating that write witnesses provide the guarantees of the read committed isolation level when all participating stores also provide the guarantees of the read committed isolation level.

We first observe that transactions which perform writes in a single phase trivially satisfy read committed, as their underlying stores provide this guarantee. We therefore consider an arbitrary sequence of writes $w_1 < \cdots < w_n$ carried out by the same transaction $\mathcal{T}_w$, but executed against distinct underlying stores. Let $\mathcal{T}_r$ be some transaction which reads some $w_n \in w_1, \ldots, w_{n-1}$. $\mathcal{T}_r$ must therefore encounter a write witness for $w_n$, and will check for the corresponding commit witness at all participating stores before proceeding. The slowest sub-transaction of $\mathcal{T}_w$, upon commit, will write the final required commit witness for $w_n$. As this witness is only written when the final commit of $\mathcal{T}_w$ occurs, we conclude that $\mathcal{T}_r$'s read itself will occur after $\mathcal{T}_w$'s commit. Thus, atomicity is preserved.

### 6.4 Read witnesses

The witness mechanism as described to this point only associates witnesses with writes and commits, guaranteeing full atomicity but not full isolation; this mechanism cannot always enforce isolation stronger than read committed [4]. Transactions which operate over snapshot-isolating ([4]) levels $\ell_1, \ell_2$, and $\ell_3$, perform reads at $\ell_1$ and $\ell_2$, and then use those reads in level $\ell_3$ may see violations of snapshot-isolation at $\ell_3$—despite all of $\ell_1, \ell_2$, and $\ell_3$ guaranteeing snapshot isolation. If $\ell_1$ permits linearizable reads, then one can recover snapshot isolation with *read witnesses*, by inserting calls to `acquire` after every read and checking for read/write witnesses before each write. We have observed few cases in which read witnesses are required; indeed none of the motivating examples for this work require them. We have not observed compelling examples in which read witnesses are impossible, as it would require multiple distinct consistency levels that are snapshot-isolated, none of which support linearizable reads. Read witnesses are currently only partially implemented in the MixT compiler.

### 6.5 Splitting and witnesses ensure mixed isolation

We return to the argument in Section 6.3. Observe that simply replaying its argument with read witnesses and reads, rather than write witnesses and writes, produces a satisfying argument for isolation; any potential violation of isolation must involve a write $w_{new}$ which occurs during some transaction which has already read $w_{old}$; but as this transaction acquired a read witness on $w_{old}$, the write to $w_{new}$ must be delayed until transaction commits.

We now are ready to demonstrate mixed isolation. We observe that any violation of mixed isolation must be due to a failure of mixed consistency, atomicity, or isolation. We have argued that witnesses and transaction splitting are sufficient to protect against failures of mixed consistency, atomicity, and isolation independently; we now conclude that their conjunction is sufficient to grant mixed isolation.

### 6.6 Mixed consistency and compositionality

Those familiar with reasoning about consistency models may wonder how MixT can ensure mixed consistency (Defn. 4.2) between non-compositional consistency models [27]. In general, non-compositional consistency models fail to capture client-centric notions of dependency between events from distinct systems; as no store knows the full set of actions performed by a client, no store is in a position to restrict possible orderings across all actions [49]. It is clearly unrealistic to ask stores to track events of which they are intentionally unaware. We therefore use a lightweight cross-store tracking mechanism to explicitly capture session order and causality at points where execution switches between stores.

Witnesses are the core of our cross-store tracking mechanism. Whenever a client performs a *cross-store sequence* of transactions $(t_1, w_2)$ in which some stores referenced by $w_2$ aren't used by $t_1$ (or vice-versa), if those stores provide non-compositional consistency models and $w_2$ performs writes to them, then we must enhance $w_2$. Specifically, we extend $w_2$ with extra phases corresponding to non-compositional models in $t_1$ which did not already appear in $w_2$. These new phases exist only to write a commit witness for $t_1$.

Any client which attempts to read $w_2$'s writes will find these witnesses, and thus any subsequent read at any of $t_1$'s stores will block until the matching commit witness is available. This conveys the session-order relationship between $t_1$ and $w_2$ to all stores; effectively, the commit witness causes the stores of $t_1$ to communicate the client's session order to the rest of the system.

This mechanism is sufficient to achieve the mixed-consistency property defined in Section 4.1. We show this by contradiction. If it were not sufficient, there would be an execution trace $t$ and consistency level $\ell$ such that $t \downarrow \ell \notin T_\ell$. Data annotated at $\ell$ lives on a single store which guarantees $\ell$; as $t$ does not guarantee $\ell$, we conclude $t$ is a mixed trace. Furthermore, as all stores store disjoint sets of objects, any inconsistency in $t$ must arrive from a violation of session order or observation order—there must be some sequence of events $e_w, e_{\ell'}, e_r \in t$ in which $e_w$ and $e_r$ live on the store of $\ell$, $e_{\ell'}$ lives on the store of some other level $\ell'$, and $e_r$ is erroneously sequentialized before $e_w$. But $e_w$ and $e_{\ell'}$ are in a cross-store sequence, so $e_{\ell'}$ was accompanied by a commit witness at $e_w$'s store. Therefore, the session executing $e_r$ must have previously read this commit witness, forcing $e_r$ to sequentialize after $e_w$.

## 7 UPGRADING CONSISTENCY

### 7.1 Semantics and Motivation

As described up to this point, MixT transactions maintain a strict separation between consistency levels; it is impossible to use a weakly consistent value to influence a strongly consistent operation. This rigid separation provides strong semantic guarantees, but can be limiting.

```
  if ((a.inbox.strong_read().size() >= 1000000 &&
       b.inbox.strong_read().size() < 1000000)
        .endorse(strong)) {
    a.declare_winner()
} else
  if ((a.inbox.strong_read().size() < 1000000 &&
       b.inbox.strong_read().size() >= 1000000)
        .endorse(strong)) {
    b.declare_winner()
}
```

Fig. 14. Strongly consistent contest logic with endorsement.

```
if (((a.inbox.size() >= 1000000 &&
      b.inbox.size() < 1000000)
  && (a.inbox.strong_read().size() >= 1000000 &&
      b.inbox.strong_read().size() < 1000000))
       .endorse(strong)) {
    a.declare_winner()
} else
if (((a.inbox.size() < 1000000 &&
      b.inbox.size() >= 1000000)
  && (a.inbox.strong_read().size() < 1000000 &&
      b.inbox.strong_read().size() >= 1000000))
       .endorse(strong)){
    b.declare_winner()
  }
```

Fig. 15. Efficient contest logic with endorsement. The programmer has introduced an additional check involving a rare strong read for when the contest is believed to be over. We also must endorse the enclosing conditional, as it still creates an indirect flow to `declare_winner`.

As an example, consider the mail-delivery example from Figure 2. During the contest, mail is delivered with causal consistency; users who view their inboxes may see a slightly stale view of the mail they've received—an acceptable semantics given the already variable latency of mail delivery itself. The system, however, needs to perform a strongly consistent read to determine a winner once the contest has finished. Let's imagine that the store supports such an operation, and exposes it to MixT via the name `strong_read`.

In MixT, the result of any expression can be declared to have an arbitrary consistency via the built-in operation `endorse(`*label*`)`. It behaves as a type-cast; the MixT compiler makes no effort to ensure that endorsements are used appropriately, as their validity often depends on complex system properties not visible to the compiler. Like all of weak consistency, endorsements must be used with care.

Using our hypothetical `strong_read` operation and MixT's `endorse` keyword, we can fix our transaction (Figure 14). This code is correct, but runs all operations at a strong consistency level—exactly what we were trying to avoid. To improve performance, we can guard each strongly consistent read with a preliminary weakly consistent test, restoring causal execution for most transactions while ensuring the declaration of a winner is still guarded by a strongly consistent

$\ell \in$ **Label**

$e ::= \ldots \mid e.\texttt{endorse}(\ell) \mid e.\texttt{ensure}(\ell)$

Fig. 16. Syntax extensions for endorsement

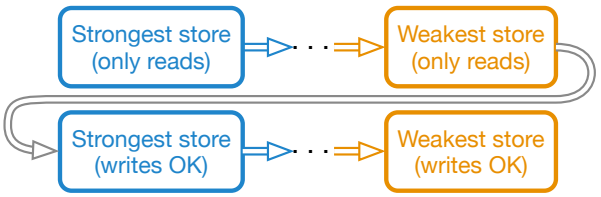


Fig. 17. Transaction phases with endorsement.

condition (Figure 15). The resulting consistency of the strong `declare_winner()` operation is no longer separable from the causally consistent read; we have accepted the possibility that our winner may be declared late.

### 7.2 Compiling Endorsements

Because of transaction splitting, endorsement is not straightforward. It fundamentally involves running weaker commands before stronger ones — and thus requires more than one phase per underlying store. But naively adding additional phases will not provide acceptable semantics; having one phase per underlying store is key to MixT's isolation guarantees.

We address these concerns with two mechanisms: *read-only phases* and read witnesses. As in other information-flow languages, endorsement is indicated by annotating the endorsed expression(Figure 16). The compiler separates the transaction into two parts: the pre-endorse part and the post-endorse part. To ensure atomicity, the pre-endorse part of the transaction is checked to ensure that it contains no writes. The code is then split into phases in the usual way, except that an artificial "pre-endorse label" is first joined with all labels in the pre-endorse code so that pre-endorse code appears to the compiler to have stronger consistency than all post-endorse code. This process is depicted in Figure 17. If full isolation is required, read witnesses (Section 6.4) may be employed. An optimization which uses read validation in place of read witnesses (as in optimistic concurrency control) is also possible [36].

## 8 IMPLEMENTATION

MixT has been implemented as four separate C++17 components, numbering almost 30,000 lines in total: the transactions language compiler (10k), the core library (2.8k), the tracking mechanism (1k), the Postgres implementation (1.4k), and support utilities (14k). The compiler can be further broken down into various phases: parsing (1.4k), A-normal transformation (500), type inference (1k), label inference (1.7k), endorsement (250), splitting (1.6k), optimization (1k), and codegen (1.4k). The current implementation supports an unbounded number of backing stores in a single application.

To evaluate MixT, we also developed several sample backing stores, operating either in-memory or based on PostgreSQL 9.4. These interfaces expose a selected set of prepared statements as custom operations and are designed to provide linearizability (with strict serializability), causal consistency (with snapshot isolation), and eventual consistency (with read-uncommitted isolation.)

### 8.1 Efficiency Optimizations

Recall that mixed-consistency transactions use witnesses to ensure atomicity, wherein an extra read operation accompanies each stated read, and an extra write operation accompanies each transaction. As described so far, this mechanism would frequently encounter stale values, harming performance for no gain in safety. Additionally, commit witnesses would slowly accumulate on the store, wasting storage. Ideally, we would be able to determine whether a value was stable across an entire store, and therefore did not require such defensive tracking behavior. To accomplish this we observe that, in order to maintain its own guarantees regarding operation order, our non-compositional store likely has some notion of a timestamp or version number already available for internal use. This assumption proves true in practice: systems such as COPS, Eiger, 2-master PostgreSQL, Bolt-on, TARDIS, HBase, and Mongo (via Vermongo) all either use these vector clocks directly or are easily modified to employ them [3, 14, 41, 42, 48]. We enhanced our `Handle` API to allow client stores to expose this notion of time to MixT through an optional `timestamp` method . We augment read and write witnesses to include the "current time" of transaction commit, and provide a lightweight TCP protocol through which backing stores can notify MixT clients of the most recent version number which is guaranteed to have reached the entire store. If the ability to access an accurate transaction commit time from within a transaction does not exist, commit witnesses can be augmented to include the addresses of all read and write witnesses created during the transaction. Leveraging this additional information, MixT avoids generating witnesses when the objects involved are already widely available, and can safely remove stale commit witnesses.

### 8.2 Remote Execution in MixT

With MixT's ability to split transactions into phases comes the opportunity to distribute the transaction code itself. By deploying a lightweight worker process alongside existing backing stores, MixT application programmers can run transaction phases directly at stores, incurring only a single round trip to establish each phase and collect its results – and allowing all witness checks to be carried out locally. In fact, this decision—to ship transaction code directly to the storage system—has become increasingly popular among high-performance data storage systems and is central to some modern databases [18, 19, 32, 56]. MixT's approach to remote execution is straightforward. We assume that each application manages its own lightweight worker at the storage location; we leave the task of ensuring code is up-to-date to the MixT application programmer.

### 8.3 MixT Compiler Implementation

We implemented MixT as a domain-specific language embedded into modern C++. MixT is written in pure C++17, and can be compiled using any C++17-compliant compiler[2]. Our entire compiler is written in constexpr C++ [45], allowing it to run during the "template expansion" step of compilation of the surrounding C++ code. Specifically, `mixt_`⟨*keyword*⟩ macros convert their arguments into a compile-time string, which is then parsed and compiled by our compiler. In order to link names within the transactions language to native C++ objects, the macros capture both the type of their arguments and their string representations, using these during the transaction compilation. All compilation, including transaction splitting, is accomplished alongside C++ compilation; none is deferred until run time. Transactions are compiled to a set of inlined, statically bound functions which are invoked from a single point in code, allowing the C++ compiler to optimize away all function-call overhead, producing machine code quite close to the syntax specified by the transaction. This approach allows MixT to support arbitrary syntax, semantics, and type systems, without requiring an external compiler or preprocessor, and without adding unnecessary run-time

[2]The MixT compiler is tested under ≥clang-3.9 and ≥g++-7.1; certain syntax extensions require -fconcepts

overhead. We are not bound to the syntax, semantics or keywords of C++; MixT's similarity to C++ is a conscious design choice. We follow the language-as-a-library paradigm: as MixT effectively adds extra phases to existing C++ compilation, to use MixT in existing C++ projects all one must do is `#include` the MixT header files.

## 9 EVALUATION

We use the MixT implementation to model an intended application domain: user-facing application servers that share one linearizable and one causally consistent underlying storage system, where application servers are geographically close to only the causal replica they are using. 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 typically housed within a single data center because latencies encountered during replication have an outsized impact on overall performance.

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

- Do mixed-consistency transactions, as promised, offer better performance than running similarly atomic transactions with strong consistency?
- What overhead is added by the witness mechanism used to preserve consistency guarantees when non-compositional consistency levels are combined?
- On what workloads does this mechanism work well? What is the performance impact of different mixtures of mixed and pure transactions?

### 9.1 Experimental Setup

To measure the performance of MixT, we simulate a geo-replicated application. In our setup, logically separate application servers each maintain connections to causally consistent and serializable databases. Connections to the serializable database experience a round-trip latency of 85ms ± 10ms; connections to causally consistent databases experience a round-trip latency of 1ms. Latency to the causal system was set by measuring ping times between an Internet2-connected university and its nearest data center; latency to the linearizable system was set by measuring ping times between Internet2-connected universities on the east and west coasts of the United States. All latency simulations are provided by the netem kernel module on Linux 3.17. We employ three separate physical machines: one hosting all clients, one hosting the causal store, and one hosting the linearizable store.

For driving load to application servers, we adopted a semi-open world model, with delay between events following an exponential distribution. We increase load by increasing the number of MixT clients, not by increasing the rate of events issued by each client.

*Using PostgreSQL as a backing store.* Our causal and linearizable stores are both backed by PostgreSQL 9.4 running on dedicated machines. These 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, SQL-specific performance concerns such as query optimization or parse time should not significantly affect 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.

To configure PostgreSQL as a causally consistent store, we created four replicas of data, and partitioned client programs among the four copies. Each instance runs transactions with snapshot isolation enforcing the guarantees of causal consistency. To order operations occurring at distinct replicas, we use a vector clock as a per-row version number. Vector clock entries are just the microsecond-resolution time at each master, so vector clock maintenance does not add serialization conflicts.

A stored PL/pgSQL operation updates these version numbers upon row modification.

These mechanisms were implemented in 1,000 lines of C++ and about 100 lines of PL/pgSQL.

Snapshot Isolation enforces the guarantees of causal consistency because 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. When reading data, a custom PL/pgSQL stored procedure automatically merges all four replicas and produces the appropriate vector clock.

### 9.2 Benchmarks

We could find no existing benchmarks for mixed-consistency transactional systems. Instead, we developed two new benchmarks intended to represent the emerging mixed-consistency landscape. The first is a simple microbenchmark based on incrementing integral counters. It features read-only transactions that fetch the value at a counter, and read-write transactions which increment that value. 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. These objects are duplicated on both a linearizable and casual store. In this benchmark, clients randomly move between causal mode, where all transactions are causally-consistent, and a linearizable mode, where all transactions are linearizable, with a fixed probability. We extend this benchmark to involve mixed-consistency transactions in the next section.

The second benchmark is the Message Groups example discussed in Section 2.2. This benchmark features four more-complex transactions: message delivery (Figure 4), user creation, inbox checking, and group joining. User creation and inbox checking are causally consistent, while message posting and group joining are mixed-consistency transactions. Each client is assigned a range of inboxes and groups from which it selects uniformly at random.

### 9.3 Counter Results

The counters benchmark offers several tuning parameters for exploring the space of workloads. As copies of our complete set of objects exist on both a causal and linearizable store, we can fine-tune both the mixture of causal and linearizable operations and the combination of reads and writes in our tests.

*Speedup Relative to Linearizability.* The most important question for MixT performance is whether mixed-consistency transactions offer a speedup compared to the simple alternative of running transactions entirely with linearizability. Figure 18 shows that, indeed, mixing causal and serializable operations considerably increases maximum throughput.

Because of the high latencies incurred by serializable transactions, increasing the causal percentage of overall operations yields significant performance improvements. These benefits level off at

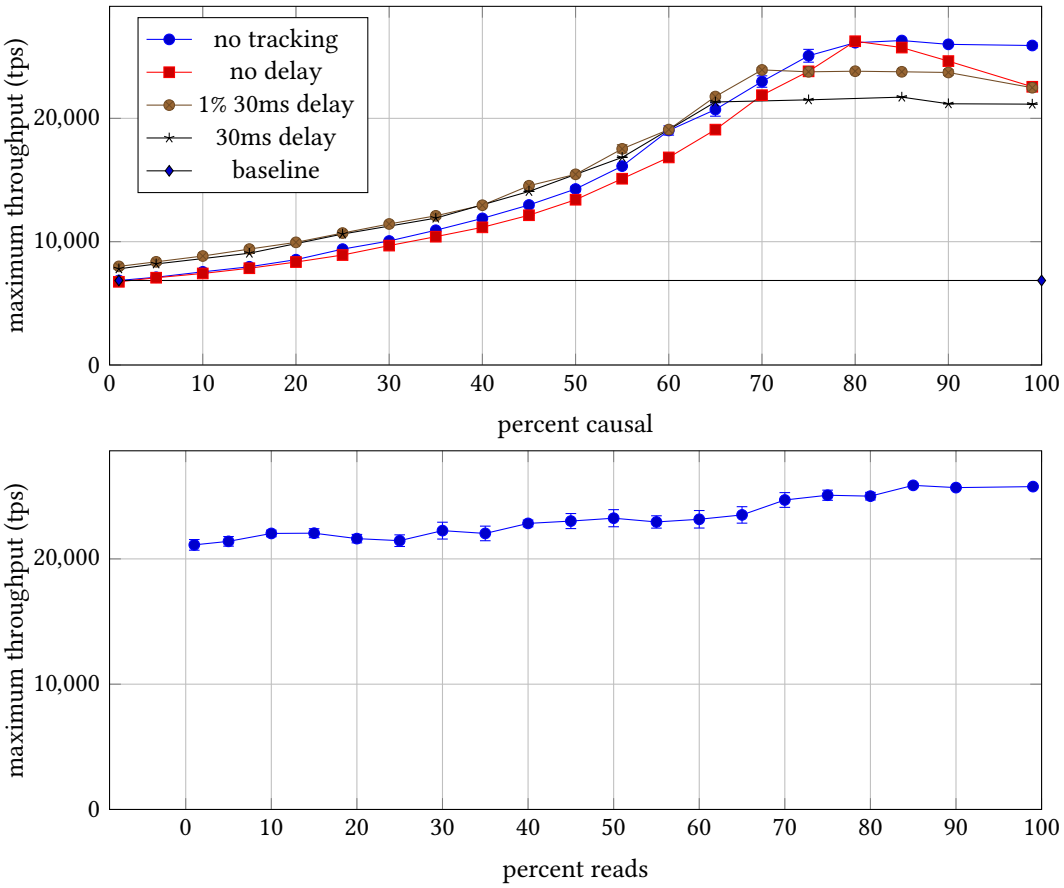


Fig. 18. Maximum throughput as a function of linearizable mix for a 70% read workload. The blue (top circle) series shows maximum achievable throughput in transactions per second (tps) without witnesses; the remaining series shows full witness tracking with progressive artificial latency. The solid black line marks 0% causal without tracking (also the leftmost blue point), which serves as a baseline. (b) Maximum throughput as a function of read share for a 75% causal workload without tracking.

about 80% causal in our tests; at this point, the causal storage system becomes overloaded, limiting the benefits of lower latency.

*Overhead of Witnesses.* One concern about MixT might be the overhead introduced by witnesses. We modify our simple counter increment test to explicitly include both linearizable and causal phases in transactions which follow a consistency mode switch, and to force witness generation in these transactions even if they would not normally require it. The rationale for this is based on the non-compositionality of causal consistency, discussed in our technical report [46]. We additionally modify our experimental setup to simulate latency of replication, first forcing approximately 1% of causal witness verifications to delay for 30ms, then explicitly delaying all causal witness verification requests by 30ms. As seen in Figure 18, the witness mechanism has a noticeable impact mostly above 60% causal, with a maximum slowdown of approximately 10%: well above the performance possible were the entire transaction mix to remain linearizable. Further demonstrating that round-trip

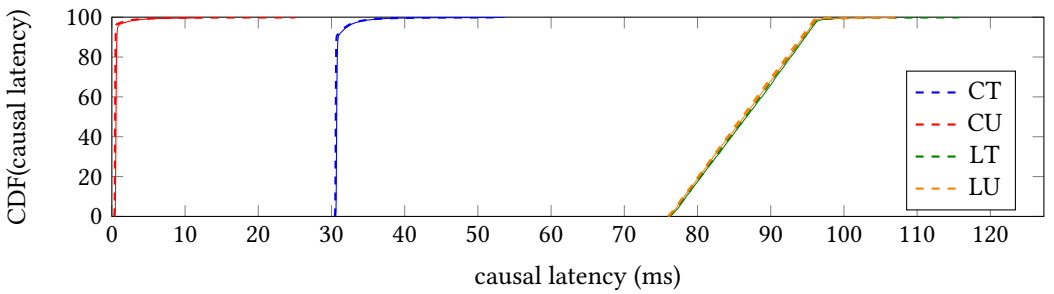


Fig. 19. CDF plots for operation latency. C: Causal, L: Linearizable, T: Tracked, U: Untracked. Dashed lines: reads, solid lines: writes. All linearizable lines appear atop each other on the right.

time is paramount, read-only transactions achieved only 20% higher maximum throughput than read-write transactions.

*Latency.* Witnesses do affect latency, especially because of design decisions made for backward compatibility. Figure 19 shows this effect. Latencies are presented as a CDF collected from the system running at 60% of maximum throughput, in a configuration in which 75% of all operations are reads and 75% of all operations use the causal store. To see the worst-case impact of witnesses, we ran this test twice: once with witnesses enabled and a forced 30ms delay ("tracked"), and once without ("untracked"). The red (leftmost) and orange (rightmost) lines on this graph measure performance without witnesses; the green (also rightmost) and blue (middle) lines represent performance with witnesses. The forced 30ms delay on witnesses is quite clear for causal operations, but there is almost no other overhead. On the other hand, the impact of witnesses on linearizable operations is negligible; as witnesses incur no replication delay in the linearizable store, they never delay linearizable phases.

In all configurations, a very small number of requests (less than .01%) experienced extreme latency—as high as 30s in the worst case. We believe this to be an artifact of unfairness in Postgres's contention management.

### 9.4 Message Groups Results

To evaluate the running Message Groups example, we use a configuration with 40,000 groups, each of which contains a single distinct user. Each of these 40,000 users has a single message in their inbox. We disable message logging, eliminating all eventually-consistent phases and leaving only causal and linearizable phases. We first run each Message Groups transaction in isolation against this initial configuration, establishing an average maximum throughput over at least 3 runs (Figure 21). This table lists the average maximum throughput for each transaction in isolation, along with the number of read and write operations executed during these transactions. For all transactions, we report the number of operations executed when in our initial configuration; message delivery and inbox downloading require more operations as the group and inbox sizes grow. The purely causal inbox download transaction benefits from the speed of causal consistency, while the mixed-consistency transactions all achieve reasonable performance despite the overhead of contacting a distant linearizable store.

We also evaluate performance on a mix of transactions: 56% inbox checking, 20% message posting, 18% group joining, and 6% user creation. We evaluate the system for 3 minutes, slowly increasing the client request rate from 2,000 tps to 5,000 tps. Average maximum throughput over 4 trials was 4,237 ± 10.5 tps with an abort rate between .0161% and .0187%. This represents a speedup of 3.5

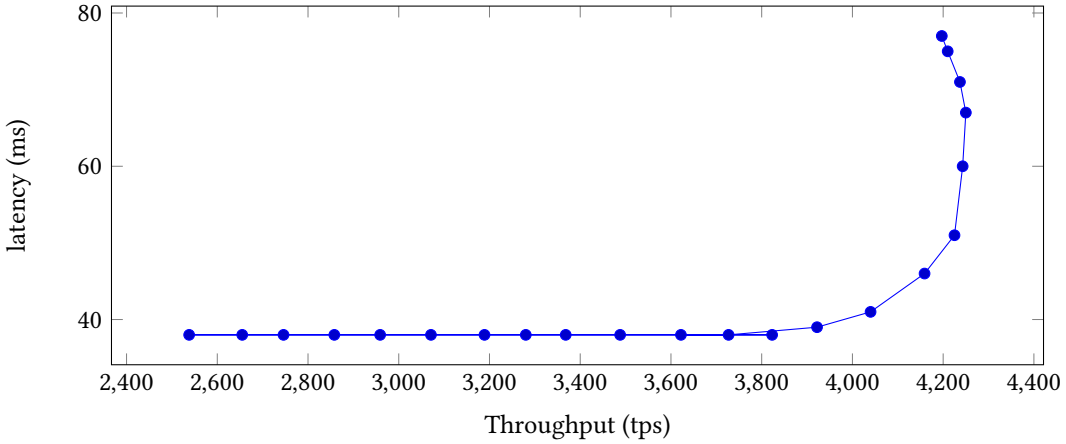


Fig. 20. Throughput vs. latency for Message Groups

| Transaction | Throughput (tps) | R | W | C | L |
|---|---|---|---|---|---|
| Check inbox | 10,626 ± 15 | 6 | 0 | ✓ | × |
| Join group | 5,430 ± 30 | 2 | 1 | ✓ | ✓ |
| Deliver message | 3,313 ± 4 | 6 | 3 | ✓ | ✓ |
| Create user | 972 ± 19 | 5 | 2 | ✓ | ✓ |

Fig. 21. Maximum throughputs for Message Groups, with standard error. The rightmost four columns give the number of reads (R) and writes (W) and indicate whether the transaction involves a causal (C) or linearizable (L) phase.

over a baseline in which all operations execute against the linearizable store (average maximum throughput: 1,228 ± 15 tps). As expected, mixed-consistency transactions yield significant speedup.

## 10 RELATED WORK

Quelea [54] and Disciplined Inconsistency [28] are the work closest to MixT in spirit. Both use user-provided data annotations to infer appropriate consistency levels for operations within a single program. The system then automatically chooses an appropriate consistency model for each operation. The Quelea approach, using Cassandra to store compressed logs of all system events, differs markedly from MixT's approach of transaction partitioning based on static information flow analysis. Disciplined Inconsistency also uses information flow to enforce separation between consistency labels but does not offer any transactional mechanism. All three systems solve distinct slices of distributed, mixed-consistency programming, suggesting a combination of approaches is an avenue for future work.

*Choosing Consistency Levels.* Choosing appropriate consistency models for data is a problem orthogonal to our work. Prior work [23, 26, 28, 29, 38, 39, 54] provide languages of constraints to describe data invariants, in turn providing the weakest consistency possible while still satisfying those constraints. Other work [8, 23, 31] aims for formal methods for users to reason about their choice of consistency level and to prove that desired code invariants are satisfied.

*Transactions in Weak Geo-Replicated Systems.* Existing work in the shared memory [16] and distributed systems [2, 17, 30, 42, 44, 53] communities has attempted to provide single-store transactions in the presence of weak consistency guarantees. This prior work focuses on definitions and mechanisms for weak transactions at a single consistency level, and indeed, we rely on the guarantees they provide.

*Mixed-Consistency Systems.* Many existing data stores provide operations with a variety of consistency guarantees [12, 15, 37, 39, 48], but without providing any semantic guarantees across operations. Others provide tools to tune consistency based primarily on performance considerations [10, 57, 61]. Guerraoui et al. [24] define a unique programming model by which programs are first presented with weakly consistent data and may choose to wait for strong data instead. These systems provide neither general transaction mechanisms nor strong semantic guarantees.

*Mixed-Consistency Systems with Transactions.* Previous work [20, 34, 35, 39, 61] focuses on progressively weakening transaction isolation based on a combination of run-time and static analysis, with the aim of enforcing strong consistency. Several papers provide mechanisms for users to choose transaction isolation levels [8, 31, 59], but do not handle the semantic anomalies involved. A few systems [16, 61] provide distributed transactions at multiple consistency levels, but allow unsafe mixing of consistency levels. Microsoft's new database Cosmos DB is a recent example, providing transactions with a choice of four well-defined consistency levels [21]. Some prior systems do enable programmers to mix transactions of different consistency with strong guarantees [39, 52, 60]. However, this line of work relies on a closed-transaction model wherein the system is aware of all possible transactions any client will run; performance is brittle because changing a single transaction somewhere in the system can significantly affect the performance of unrelated transactions. This work cannot mix consistency within a single transaction, and it focuses on a single store. Nevertheless, these systems could be used by MixT as backing stores.

*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 [25]. 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.

*Enhancing Consistency of Existing Systems.* Beyond SQL, some existing work has focused on mechanisms that upgrade the consistency guarantees of weakly consistent underlying stores [3, 30, 53]. Indeed, several projects [42, 54] use this approach internally, adding consistency layers atop existing distributed systems like Cassandra.

## 11 CONCLUSION

We have introduced a new domain-specific programming language for writing modern geodistributed applications that need to trade off performance and consistency. The mixed-consistency transactions offered by MixT make it possible for programmers to safely combine data from multiple consistency levels in the same transaction, with confidence that weaker data does not corrupt the guarantees of stronger data. Appealingly, this model can be implemented in a backward-compatible way on top of existing stores that offer their own distinct consistency guarantees, without disrupting

legacy applications on those stores. The performance results suggest that for geodistributed applications, mixed-consistency transactions enable higher performance by using weaker consistency models selectively and safely.

## ACKNOWLEDGMENTS

The authors would like to thank Jane Brown, Stephanie DeMane, Jon Sailor, Cody Mello, and Robert Mustacchi for their help determining industry applications. The authors would like to thank Ethan Cecchetti for help with PostgreSQL, Nate Nystrom for discussions about information flow and consistency, Isaac Sheff for extensive consultation, and Tom Magrino, Daoji Huang, Eric Lee, Eston Schweickart, Andrew Hirsch, and Sydney Zink for their editorial assistance.

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, and by NSF grant 1717554.

## REFERENCES

[1] A. Adya. 1999. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions.* Ph.D. Dissertation. Massachusetts Institute of Technology, Cambridge, MA.

[2] Masoud Saeida Ardekani, Pierre Sutra, and Marc Shapiro. 2013. Non-Monotonic Snapshot Isolation: Scalable and Strong Consistency for Geo-Replicated Transactional Systems. In *Reliable Distributed Systems (SRDS), 2013 IEEE 32nd International Symposium on*. IEEE, 163–172.

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

[4] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*.

[5] K. J. Biba. 1977. *Integrity Considerations for Secure Computer Systems.* Technical Report ESD-TR-76-372. USAF Electronic Systems Division, Bedford, MA. (Also available through National Technical Information Service, Springfield Va., NTIS AD-A039324.).

[6] Eric Brewer. 2010. A Certain Freedom: Thoughts on the CAP Theorem. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*. ACM, 335–335.

[7] Jane W S Brown. 2017. Personal communication. (Nov. 2017). Google, Inc.

[8] Lucas Brutschy, Dimitar Dimitrov, Peter Müller, and Martin T. Vechev. 2017. Serializability for Eventual Consistency: Criterion, Analysis, and Applications. In *44$^{th}$ ACM Symp. on Principles of Programming Languages (POPL)*. 458–472.

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

[10] Shankha Chatterjee and Wojciech Golab. 2017. Self-Tuning Eventually-Consistent Data Stores. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems*. Springer, 78–92.

[11] Alvin Cheung, Samuel Madden, Owen Arden, and Andrew C. Myers. 2012. Automatic Partitioning of Database Applications. *PVLDB* 5, 11 (Aug. 2012), 1471–1482.

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

[13] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 8.

[14] Natacha Crooks, Youer Pu, Nancy Estrada, Trinabh Gupta, Lorenzo Alvisi, and Allen Clement. 2016. TARDiS: A Branch-and-Merge Approach To Weak Consistency. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 1615–1628.

[15] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key–Value Store. In *21$^{st}$ ACM Symp. on Operating System Principles (SOSP)*.

[16] Brijesh Dongol, Radha Jagadeesan, and James Riely. 2018. Transactions in Relaxed Memory Architectures. In *45$^{th}$ ACM Symp. on Principles of Programming Languages (POPL)*. 18:1–18:29.

[17] Jiaqing Du, Călin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. 2014. Gentlerain: Cheap and Scalable Causal Consistency with Physical Clocks. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 1–13.

[18] Ayush Dubey, Greg D. Hill, Robert Escriva, and Emin Gün Sirer. 2016. Weaver: A High-Performance, Transactional Graph Database Based on Refinable Timestamps. *Proceedings of the VLDB Endowment* 9, 11 (2016), 852–863.

[19] Jennie Duggan, Aaron J Elmore, Michael Stonebraker, Magda Balazinska, Bill Howe, Jeremy Kepner, Sam Madden, David Maier, Tim Mattson, and Stan Zdonik. 2015. The BigDAWG Polystore System. *ACM SIGMOD Record* 44, 2 (2015), 11–16.

[20] Lei Gao, Mike Dahlin, Amol Nayate, Jiandan Zheng, and Arun Iyengar. 2003. Application-Specific Data Replication for Edge Services. In *Proceedings of the 12th international conference on World Wide Web*. ACM, 449–460.

[21] Mimi Gentz, Aravind Krishna R, Luis Bosquez, Mark McGee, Tyson Nevil, Kris Crider, Yaron Y. Goland, Andy Pasic, and Ji Huang Carol Zeumault. 2017. Welcome to Azure Cosmos DB. https://docs.microsoft.com/en-us/azure/cosmos-db/introduction. (2017).

[22] Joseph A. Goguen and Jose Meseguer. 1982. Security Policies and Security Models. In *IEEE Symp. on Security and Privacy*. 11–20.

[23] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 2016. 'Cause I'm Strong Enough: Reasoning About Consistency Choices in Distributed Systems. In *43rd ACM Symp. on Principles of Programming Languages (POPL)*. 371–384.

[24] Rachid Guerraoui, Matej Pavlovic, and Dragos-Adrian Seredinschi. 2016. Incremental Consistency Guarantees for Replicated Objects. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. 169–184.

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

[26] M. Herlihy. 1991. Wait-Free Synchronization. *ACM Transactions on Programming Languages* 1, 13 (Jan 1991), 124–149.

[27] M. Herlihy and J. Wing. 1988. *Linearizability: A Correctness Condition for Concurrent Objects*. Technical Report CMU-CS-88-120. Carnegie Mellon University, Pittsburgh, Pa.

[28] Brandon Holt, James Bornholt, Irene Zhang, Dan Ports, Mark Oskin, and Luis Ceze. 2016. Disciplined Inconsistency with Consistency Types. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*. ACM, 279–293.

[29] Brandon Holt, Irene Zhang, Dan Ports, Mark Oskin, and Luis Ceze. 2015. Claret: Using Data Types for Highly Concurrent Distributed Transactions. In *1st Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC)*. Article 4, 4:1–4:4 pages.

[30] Ta-Yuan Hsu and Ajay D. Kshemkalyani. 2018. Value the Recent Past: Approximate Causal Consistency for Partially Replicated Systems. *IEEE Transactions on Parallel and Distributed Systems* 29, 1 (2018), 212–225.

[31] Gowtham Kaki, Kartik Nagar, Mahsa Nazafzadeh, and Suresh Jagannathan. 2017. Alone Together: Compositional Reasoning and Inference for Weak Isolation. *arXiv preprint arXiv:1710.09844* (2017).

[32] 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. 2008. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *PVLDB* 1, 2 (Aug. 2008).

[33] Rusty Klophaus. 2010. Riak Core: Building Distributed Applications Without Shared State. In *ACM SIGPLAN Commercial Users of Functional Programming (CUFP '10)*. ACM, New York, NY, USA, Article 14, 1 pages.

[34] Tim Kraska, Martin Hentschel, Gustavo Alonso, and Donald Kossmann. 2009. Consistency Rationing in the Cloud: Pay Only When it Matters. *Proceedings of the VLDB Endowment* 2, 1 (2009), 253–264.

[35] Tim Kraska, Gene Pang, Michael J Franklin, Samuel Madden, and Alan Fekete. 2013. MDCC: Multi-Data Center Consistency. In *Proc. 8th ACM European Conference on Computer Systems*. ACM, 113–126.

[36] H. T. Kung and J. T. Robinson. 1981. On Optimistic Methods for Concurrency Control. *ACM Trans. on Database Systems* 6, 2 (June 1981), 213–226.

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

[38] Cheng Li, Joao Leitao, Allen Clement, Nuno Preguiça, Rodrigo Rodrigues, and Viktor Vafeiadis. 2014. Automating the Choice of Consistency Levels in Replicated Systems. In *USENIX Annual Technical Conference*.

[39] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. 2012. Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary. In *10th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*.

[40] Jed Liu and Andrew C. Myers. 2014. Defining and Enforcing Referential Security. In *3rd Conf. on Principles of Security and Trust (POST)*. 199–219.

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

[42] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. 2013. Stronger Semantics for Low-Latency Geo-Replicated Storage. In *10th USENIX Symp. on Networked Systems Design and Implementation (NSDI)*. 313–328.

[43] Daryl McCullough. 1987. Specifications for Multi-Level Security and a Hook-up Property. In *IEEE Symp. on Security and Privacy*. IEEE Press.

[44] Syed Akbar Mehdi, Cody Littley, Natacha Crooks, Lorenzo Alvisi, Nathan Bronson, and Wyatt Lloyd. 2017. I Can't Believe It's Not Causal! Scalable Causal Consistency with No Slowdown Cascades.. In *NSDI*. 453–468.

[45] Scott Meyers. 2014. *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++ 11 and C++ 14*. O'Reilly Media, Inc..

[46] Matthew Milano and Andrew Myers. 2017. MixT: A Language for Mixing Consistency in Geodistributed Transactions: Technical Report. (2017).

[47] C. H. Papadimitriou. 1979. The Serializability of Concurrent Database Updates. *J. ACM* 26, 4 (Oct. 1979), 631–653.

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

[49] Jelica Protic, Milo Tomasevic, and Veljko Milutinovic. 1996. Distributed Shared Memory: Concepts and Systems. *IEEE Parallel & Distributed Technology: Systems & Applications* 4, 2 (1996), 63–71.

[50] Andrei Sabelfeld and Andrew C. Myers. 2003. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications* 21, 1 (Jan. 2003), 5–19.

[51] Amr Sabry and Matthias Felleisen. 1993. Reasoning About Programs in Continuation-Passing Style. *Lisp and Symbolic Computation* 6, 3–4 (Nov. 1993), 289–360.

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

[53] Kazuyuki Shudo and Takashi Yaguchi. 2017. Causal Consistency for Data Stores and Applications as They are. *Journal of Information Processing* 25 (2017), 775–782.

[54] K. C. Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. 2015. Declarative Programming over Eventually Consistent Data Stores. In *36$^{th}$ ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*.

[55] Geoffrey Smith and Dennis Volpano. 1998. Secure Information Flow in a Multi-Threaded Imperative Language. In *25$^{th}$ ACM Symp. on Principles of Programming Languages (POPL)*. 355–364.

[56] Michael Stonebraker and Ariel Weisberg. 2013. The VoltDB Main Memory DBMS. *IEEE Data Eng. Bull.* 36, 2 (2013), 21–27.

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

[58] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, and Mike J. Spreitzer. 1995. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *15$^{th}$ ACM Symp. on Operating System Principles (SOSP)*. 172–183.

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

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

[61] Yingyi Yang, Yi You, and Bochuan Gu. 2017. A Hierarchical Framework with Consistency Trade-off Strategies for Big Data Management. In *Computational Science and Engineering (CSE) and Embedded and Ubiquitous Computing (EUC), 2017 IEEE International Conference on*, Vol. 1. IEEE, 183–190.

[62] Haifeng Yu and Amin Vahdat. 2000. Design and Evaluation of a Continuous Consistency Model for Replicated Services. In *4$^{th}$ USENIX Symp. on Operating Systems Design and Implementation (OSDI)*.

[63] Steve Zdancewic and Andrew C. Myers. 2003. Observational Determinism for Concurrent Program Security. In *16$^{th}$ IEEE Computer Security Foundations Workshop (CSFW)*. 29–43.

[64] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. 2002. Secure Program Partitioning. *ACM Trans. on Computer Systems* 20, 3 (Aug. 2002), 283–328.