TARDiS: A Branch-and-Merge Approach To Weak Consistency

Natacha Crooks[‡] Youer Pu[‡] Nancy Estrada[†]
Trinabh Gupta[‡] Lorenzo Alvisi[‡] Allen Clement[§]
[‡]The University of Texas at Austin [†]MPI-SWS [§]Google,inc.

ABSTRACT

This paper presents the design, implementation, and evaluation of TARDiS (**T**ransactional **A**synchronously **R**eplicated **Di**vergent **S**tore), a transactional key-value store explicitly designed for weakly-consistent systems. Reasoning about these systems is hard, as neither causal consistency nor per-object eventual convergence allow applications to deal satisfactorily with write-write conflicts. TARDiS instead exposes as its fundamental abstraction the set of conflicting branches that arise in weakly-consistent systems. To this end, TARDiS introduces a new concurrency control mechanism: branch-on-conflict. On the one hand, TARDiS guarantees that storage will appear sequential to any thread of execution that extends a branch, keeping application logic simple. On the other, TARDiS provides applications, when needed, with the tools and context necessary to merge branches atomically, when and how applications want. Since branch-on-conflict in TARDiS is fast, weakly-consistent applications can benefit from adopting this paradigm not only for operations issued by different sites, but also, when appropriate, for conflicting local operations. We find that TARDiS reduces coding complexity for these applications and that judicious branch-on-conflict can improve their local throughput at each site by two to eight times.

1. INTRODUCTION

This paper describes the design, implementation, and evaluation of TARDiS, an asynchronously replicated, multi-master, transactional key-value store designed for applications built above weakly-consistent and often non-transactional systems, such as Riak [10], MongoDB [37], or Cassandra [6]. TARDiS renounces the one-size-fits-all abstraction of sequential storage and instead exposes applications, when appropriate, to concurrency and distribution. This unconventional design is predicated on a simple notion: to help developers resolve the anomalies that arise in such applications, *each replica should faithfully store the full context necessary to understand how the anomalies arose in the first place, but only expose that context to applications when needed.*

In light of the CAP theorem [14, 23], many wide-area services and applications [16] choose to renounce strong consistency and focus instead on providing the ALPS properties [31] of **A**vailability, low **L**atency, **P**artition tolerance, and high **S**calability. Distributed ALPS applications, however, are hard to reason about. Geographically distinct replicas can issue conflicting operations, and the read-write and write-write conflicts that can ultimately result from these operations may cause replicas' states to diverge.

Many prior systems have attempted to insulate applications from this complexity by relying on a combination of two techniques: *causal consistency* [4, 9, 21, 32, 44, 47] to mitigate the effects of read-write conflicts, and per-object *eventual convergence* [20, 31, 33, 34, 44, 48] to address write-write conflicts. These systems strive to keep complexity in check by aggressively preserving the familiar abstraction that an application's state evolves through a linear sequence of updates. Any perturbation to this abstraction is nipped in the bud, either within the storage layer—by enforcing per-object convergence through simple deterministic resolution policies—or by asking the application to resolve the state of objects with conflicting updates as soon as conflicts arise [20, 25, 48, 31].

These techniques, however, are not sufficient to uphold the abstraction of sequential storage in the presence of concurrent updates. Worse, causal order and per-object convergence provide no support for meaningfully resolving conflicts between concurrent sequences of updates that involve multiple objects: indeed, they often destroy information that could have helped the application address these anomalies. For example, deterministic writer-wins, a common technique to achieve convergence [31], hides write-skew from applications (§2). Similarly, exposing multivalued objects without context obscures cross-object semantic dependencies (§2).

Anomalies like write-skew are intrinsic to ALPS applications. The issue is then neither how to prevent them (one can't), nor how to resolve them transparently (application-specific knowledge is often indispensable): rather, it is how to provide ALPS applications with the best possible system support when merging conflicting states.

To this end, TARDiS deliberately abandons a strictly sequential view of storage, and instead gives applications flexibility. If all is well, storage at each replica appears sequential; when conflicts must be resolved, however, the intricate details of distribution become available. As in Git [24], users operate on their own branch and explicitly request (when convenient) to see concurrent modifications, using the history recorded by the underlying branching storage to help them resolve conflicts. Unlike Git, however, branching in TARDiS does not rely on specific user commands, but occurs implicitly, to preserve availability in the presence of conflicts, using three core mechanisms: (*i*) branch-on-conflict, (*ii*) inter-branch isolation, and (*iii*) application-driven cross-object merge.

Branch-on-conflict lets TARDiS logically fork its state whenever it detects conflicting operations, and store the conflicting branches explicitly. Inter-branch isolation guarantees that storage will appear as sequential to any thread of execution that extends a

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA,USA

DOI: http://dx.doi.org/10.1145/2882903.2882951

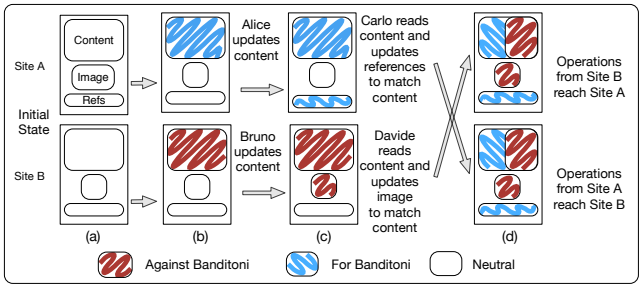


Figure 1: Weakly-consistent Wikipedia

branch, keeping application logic simple. Finally, TARDiS leaves the task of deciding if, when, and how divergent branches should be merged to the application, rather than to the storage layer, which is generally unsuited to leverage relevant semantic information.

At first glance, TARDiS' design may appear counter-intuitive: isn't a simpler abstraction, such as sequential storage, easier to reason about? Our view is that abstractions should indeed be made as simple as possible—but no simpler: a simplistic abstraction that overlooks critical context can actually make reasoning harder. Through its richer interface, TARDiS gives applications access to context that is essential to reasoning about concurrent updates, reducing the complexity of programming ALPS applications and improving their performance. It does so via the following four key properties.

TARDiS knows history. Each TARDiS site stores a DAG that records all branches generated during an execution, and uses a new algorithm, *DAG compression*, to track the minimal information needed to support branch merges. This context, which traditional storage systems hastily discard, can prove invaluable when programming ALPS applications (§7). We find, for example, that using TARDiS rather than BerkeleyDB [38] to implement CRDTs [42]—a library of scalable, weakly-consistent datatypes—cuts code size by half, improves performance by four to eight times, and reduces development time by a factor of three.

TARDiS merges branches, not objects. Prior systems that, like TARDiS, admit parallel versions of the same object [3, 20, 34, 48] have systematically taken a strictly per-object view of multiversions. With no support for enforcing the cross-object consistency demands expressed in many application invariants, such systems make conflict resolution more difficult and error prone. To efficiently construct and maintain branches, TARDiS introduces the notion of *conflict tracking*. By summarizing branches as a set of fork points and merge points, conflict tracking significantly reduces the metadata overhead experienced by many systems that enforce causal consistency [8, 21].

TARDiS is expressive. Despite exposing a different abstraction, TARDiS supports many isolation levels (serializability, snapshot isolation, read-committed [11]) and consistency guarantees (read-my-writes [46], causal consistency [4]). It does so with minimal changes to applications' code and with performance comparable to that of BerkeleyDB, a commercially available Java database. TARDiS achieves this flexibility by reformulating isolation and consistency requirements as a set of pre- and post- conditions.

TARDiS improves performance of the local site. A unique feature of TARDiS is that it allows ALPS applications to apply weak-consistency principles end-to-end, by triggering branch-on-conflict not only for operations issued by different sites, but also for locally conflicting operations. When so configured, TARDiS handles local conflicts not through abort/rollback and locking, but by logically forking the local datastore, which in our implementation is a very fast operation. Of course, this feature is not beneficial to all applications, as producing a large number of additional branches may increase merging complexity dramatically. However, we find that the ALPS applications that TARDiS targets, where weak consistency and merging are first-order concerns, can leverage this feature to increase their throughput significantly: applying this technique to Retwis [41], a commonly used Twitter-like ALPS application [39, 41, 44, 51], yields a three-fold improvement in throughput with negligible increase in complexity. TARDiS enables this speedup by extending the copy-on-write techniques present in multiversioned systems to support not just stale snapshots, but also branches.

2. THE GAP BETWEEN CAUSALITY AND REALITY

In the classic example used to illustrate the virtues of causal consistency, Alice gains assurance that Bob, whom she had defriended before posting her Spring-break photos, will not be able to access her pictures, even though Alice and Bob access the photo-sharing application using different sites [8, 17, 31]. ALPS applications, however, face another class of anomalies—write-write-conflicts—that causal consistency cannot prevent, detect, or repair.

To illustrate, consider the process of updating a Wikipedia page consisting of multiple HTML objects (Figure 1(a)). The page in our example, about a controversial politician, Mr. Banditoni, is frequently modified, and is thus replicated on two sites, A and B. Assume, for simplicity, that the page consists of just three objects—the content, references, and an image. Alice and Bruno, who respectively strongly support and strongly oppose Mr. Banditoni, concurrently modify the content section of the webpage on sites A and B to match their political views (Figure 1(b)). Carlo reads the content section on site A, which now favors Mr. Banditoni, and updates the reference section accordingly by adding links to articles that praise the politician. Similarly, Davide reads the update made by Bruno on site B and chooses to strengthen the case made by the content section by updating the image to a derogatory picture of Mr. Banditoni (Figure 1(c)). Eventually, the operations reach the other site and, although nothing in the preceding sequence of events violates causal consistency, produce the inconsistent state shown in Figure 1(d): a content section that exhibits a write-write conflict; a reference section in favor of Mr. Banditoni; and an image that is against him. Worse, there is no straightforward way for the application to detect the full extent of the inconsistency: unlike the explicit conflict in the content sections, the discrepancy between image and references is purely semantic, and would not trigger an automatic resolution procedure.

To the best of our knowledge, this scenario presents an open challenge to existing weakly-consistent systems, which exhibit at least one of the following two properties:

(*i*) *Syntactic conflict resolution.* To maintain the abstraction of sequential storage, many systems use fixed, syntactic resolution policies to reconcile write-write conflicts [31]. Deterministic writer-wins (DWW), for example, resolves write-write conflicts identically at all sites, ensuring that applications never see conflicting writes and guaranteeing eventual convergence. In our example, this policy would choose Bruno's update. However, this is not sufficient to restore consistency, as it ignores the relationship between the content, references, and images of the webpage. The datastore's greedy attempt at syntactic conflict resolution is not only inadequate to bridge this semantic gap, but leads to losing valuable information (here, Alice's update).

(*ii*) *Lack of cross-object semantics.* Some systems choose to push conflict resolution to the application [20, 48], but on a per-object

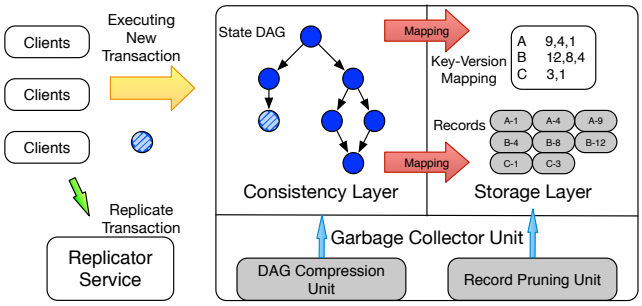


Figure 2: TARDiS architecture

basis only. Though more flexible than a purely syntactic solution, this approach, which reduces conflict resolution to the merging of explicitly conflicting writes, is still overly narrow. For example, it would not address the inconsistencies, such as the one between the references and the image, that do not produce a write-write conflict. Yet, the effects of a write-write conflict on an object do not end with that object: Carlo and Davide update references and images as they do because they have *read* the conflicting updates to the original content section. Indeed, any operation that depends on one of two conflicting updates is potentially incompatible with all the operations that depend on the other: the shockwaves from even a single write-write conflict may spread to affect the state of the entire database.

There is currently no straightforward way for applications to resolve consistently the kind of multi-object, indirect conflicts that our example illustrates. Transactions [31, 32], an obvious candidate, are powerless when the objects that directly or indirectly reflect a write-write conflict are updated, as in our example, by different users. After Bruno's update, the application has no way to know that Davide's update is forthcoming: it must therefore commit Bruno's transaction, forcing Bruno's and Davide's updates into separate transactions. Nor would it help to change the granularity of the object that defines the write-write conflict—in our example, by making that object be the entire page. It would be easy to correspondingly scale up the example, using distinct pages that link each other. Short of treating the entire database as the "object", it is futile to try to define away these inconsistencies by redrawing the objects' semantic boundaries.

3. BRIDGING THE GAP: BRANCHES

TARDiS' design is motivated by the belief that isolating ALPS applications from the harsh but inescapable reality of independent conflicting writes, and from the resolution process that they require, is a well-intentioned fallacy. TARDiS instead embraces transparency. By default, applications execute on a branch, and hence perceive storage as sequential. But when anomalies arise, TARDiS provides two novel features that simplify reconciliation.

First, it exposes applications to the resulting independent branches, and to the states at which the branches are created (*fork points*) and merged (*merge points*). Second, it supports atomic merging of conflicting branches and lets applications choose when and how to reconcile them (§3.1). These features allow TARDiS to offer ALPS applications the opportunity to pursue, down to each site's local datastore, an intriguing notion: that of turning weak consistency, through a bit of system-design judo, from a weakness to an unlikely strength (§3.2).

3.1 State branching and merging

As the discussion in Section 2 has illustrated, even a single write-write conflict has the potential to affect the entire state of a database. In essence, conflicting operations fork the entire state of the system, creating distinct *branches*, each tracking the linear evolution of the datastore according to a separate thread of execution. The Wikipedia example hence consists of two branches: one in support of Banditoni, and one against him. Elevating branches to the datastore's fundamental abstraction has two complementary advantages. First, users that operate within a given thread of execution continue to perceive the application's state as evolving linearly. Second, when it becomes necessary to alert users to the existence of concurrent updates that conflict with that linear view, branches are the natural unit of merging.

Resolving conflicts in ALPS applications often requires semantic context. Replicas, however, only see a sequence of read/write operations and are unaware of the application-level logic and invariants that relate these operations [5]. Therefore, they should avoid deterministic quick fixes, and instead give applications the information they need to decide what is best. Branches, together with their fork and merge points, naturally encapsulate such information: they make it easy to identify all the objects to be considered during merging and pinpoint when and how the conflict developed. This context can reduce the complexity and improve the efficiency of automated merging procedures, as well as help system administrators when user involvement is required. In our example, a Wikipedia moderator presented with the two conflicting branches would be able to reconstruct the events that led to them and handle the conflicting sources according to Wikipedia's guidelines [50]. Note that merging need not simply involve deleting one branch. Indeed, branching and merging states enables merging strategies with richer semantics than aborts or rollbacks [44].

3.2 Weak consistency end-to-end

Write-write conflicts in distributed systems are not restricted to remote sites: conflicting operations can also happen locally. Unlike remote conflicts, however, they are immediately detectable, and hence they are typically handled by the datastore through locking or rollback. When considering the specific nature of ALPS applications, however, two observations bring this common-sense approach into question. First, desirable as it may be, the abstraction of a sequential store cannot be preserved end-to-end: at the distributed system level, it falls apart. Second, the design complexity of having to program against the possibility of remote conflicts is already factored into ALPS applications, whose semantics often support simple merging procedures.

These observations lead us to explore an unconventional proposition: design and implement a datastore for ALPS applications with branching as its fundamental abstraction, used to model conflicts end-to-end, from the level of the distributed system down to that of local storage. This stance does not simply have an aesthetic appeal: eliminating locks and rollbacks from the performance critical path offers the potential, through a lightweight implementation of branching, to improve throughput at local sites.

Accordingly, TARDiS gives ALPS applications the option of handling local conflicts through branch-on conflict, rather than syncronization. Naturally, one must tread carefully: out-of-control branching can turn reasoning about the state of the system into a nightmare. TARDiS thus lets applications tune the degree of local branching allowed, so they can strike the balance between performance and complexity that best meets their requirements.

3.3 System Goals

The challenge is then to develop a datastore that can keep track of independent execution branches, record fork and merge points, facilitate reasoning about branches and, as appropriate, atomically

Constraint	B	E	Description
Any	√	√	Always Satisfies
Serializability		√	Guarantees Serializability
Snapshot Iso		√	Guarantees Snapshot Isolation
Read Committed		√	Guarantees Read Committed
No Branching		√	State has no children
K-Branching		√	State has fewer than k-1 children
Parent	√		State where client last committed
Ancestor	√		Child of client's last committed state
State Identifier	√		State ID matches the specified ID

Table 1: Begin (B) and end (E) constraints supported by TARDiS

merge them—while keeping performance and resource overheads comparable to those of weakly-consistent systems. Such a system should satisfy the following three requirements:

- **Simple interface** The datastore should expose an interface that allows developers to navigate and manipulate branches; that interface should minimize any increase in complexity and need to modify legacy code.
- **Good Performance** The datastore should efficiently create, track, and merge branches; its performance should match or surpass that of a storage system that is strictly sequential and does not keep track of history.
- **Minimal Space Overhead** The datastore should have a reasonable memory footprint and manage efficiently the space overhead associated with keeping multiple executions and their fork points.

The rest of this paper presents the design of TARDiS, focusing on how it satisfies these three requirements.

4. TARDiS ARCHITECTURE

The TARDiS transactional key-value store tracks conflicting execution branches using three mechanisms: *branch-on-conflict*, *inter-branch isolation*, and *application-specific merge*. TARDiS uses multi-master asynchronous replication: transactions first execute locally at a specific site, and are then asynchronously propagated to all other replicas. Each replica consists of four components: a storage layer, a consistency layer, a garbage collector unit, and a replicator service (Figure 2).

The *storage layer* stores records in a disk-backed B-Tree. Currently, every site stores a full copy of the database, though TARDiS can be extended to support data partitioning (§6.4). TARDiS is a multiversioned system: every update operation creates a new record version. The mapping between versions and keys is stored in an in-memory cache for fast traversal.

The *consistency layer* tracks branches with the help of a directed acyclic graph, whose vertices correspond to logical *states* of the datastore: each transaction that updates a record generates a new state. TARDiS' logic is geared towards efficiently mapping the consistency layer to the storage layer: when a transaction creates a new state, it extends the State DAG by appending the state to its chosen branch of execution. Any newly created record version is marked by this state's identifier. When a transaction executes a get operation, it uses information contained in the State DAG to determine which object versions are visible to its branch.

TARDiS' *garbage collector unit* comprises a DAG compression submodule and a record pruning submodule. DAG compression periodically discards intermediate states that are no longer needed, and record pruning then removes the associated object versions. Garbage collection allows TARDiS' to maintain memory and storage overheads that are comparable to traditional weakly-consistent systems that do not track history.

S	M	return type	method
√		transaction	begin(*beginConstraint*)
	√	transaction	beginMerge(*beginConstraint*)
√	√	void	put(*key, value*)
√		value	get(*key*)
	√	value	getForID(*key, StateID[]*)
	√	key[]	findConflictWrites(*StateID[]*)
	√	forkPoints[]	findForkPoints(*StateID[]*)
√	√	abort|commit	commit(*endConstraint*)

Table 2: TARDiS API - S:single mode, M:merge mode

Finally, the *replicator service* propagates committed transactions and applies remote transactions as appropriate.

5. USING TARDiS

To help weakly-consistent applications deal with the complexity of resolving the conflicts they encounter, TARDiS' API (Table 2) addresses three competing concerns: (*i*) minimizing programming complexity, (*ii*) simplifying reasoning about concurrent branches, and (*iii*) controlling the degree of local branching.

5.1 Interface

To ease programming, TARDiS can operate in either *single mode* or *merge mode*. In the default *single mode*, the programmer is allowed to (transactionally) read from and write to a single branch. Programming proceeds exactly as in traditional transactional systems, except that programmers must now select a branch to operate on. Thus, porting an application to TARDiS requires only adding a branch-selection call. Figure 3 illustrates this point by sketching the implementation of a simple counter. The two single mode operators, `increment` and `decrement`, now take as parameters predicates that specify the properties of the desired branch (more on these predicates below), but these operators are otherwise implemented exactly as one would on sequential storage.

In *merge mode*, programmers can instead explicitly reconcile conflicting branches via *merge transactions*, that read from multiple states and write back to a single, merged state. TARDiS' merge mode allows users to perform cross-object resolution atomically: this simplifies conflict resolution significantly, much like transactions simplify application logic by allowing users to modify multiple objects atomically.

When reconciling branches, applications will typically (*i*) detect conflicting objects; (*ii*) identify where branches forked; and (*iii*) determine the values of conflicting keys at this fork point (§3). To help applications obtain the information that they need for merging, TARDiS adds three new API calls: `findConflictWrites`, `findForkPoints`, and `getForID`. `findConflictWrites` returns the list of objects with conflicting values across all selected branches, freeing programmers from the need to implement application-level mechanisms for tracking what has to be resolved. `findForkPoints` returns, for a given set of states, the structured set of fork points that reveals the branching structure of the corresponding State DAG. For simplicity, in this paper we restrict ourselves to the case where `findForkPoints` returns a single fork point. Finally, `getForID` allows the application to request any object version, freeing application programmers from the need to track how the datastore evolves. We expect applications to call this function primarily to obtain object values at the fork point(s), and to use this information to resolve conflicts in an application-specific way before writing the merged value back. Though TARDiS supports concurrent merges for full flexibility, we expect that, for simplicity, most applications will restrict merging to a single site.

```
1  func increment(counter)
2   Tx t = begin(AncestorConstraint)
3   int value = t.get(counter)
4   t.put(counter, value + 1)
5   t.commit(SerializabilityConstraint)

7  func decrement(counter)
8   Tx t = begin(AncestorConstraint)
9   int value = t.get(counter)
10  t.put(counter, value - 1)
11  t.commit(SerializabilityConstraint)

13 func merge()
14   Tx t = beginMerge(AnyConstraint)
15   forkPoint forkPt =
16          t.findForkPoints(t.parents).first
17   int forkVal = t.getForID(counter,forkPt)
18   list<int> currentVals =
19          t.getForID(counter, t.parents)
20   int result = forkVal
21   foreach c in currentVals
22     result += (c - forkVal)
23   t.put(counter,result)
24   t.commit(SerializabilityConstraint)
```

Figure 3: TARDiS' counter implementation

TARDiS helps applications reason about concurrent execution branches and control the degree of local branching through *begin* and *end constraints*—predicates associated with begin and commit commands that specify which branch a transaction can execute from. Intuitively, begin constraints select what states the transaction can read, while end constraints specify what conditions must hold upon commit. TARDiS supports the constraints listed in Table 1: along with their union and intersection, they are sufficiently flexible to express traditional database isolation levels, such as serializability and snapshot isolation [11], as well as distributed-system guarantees such as read-my-writes [46]. For example, an application could use the *Ancestor* begin constraint and the union of the *Serializability* and *No Branching* end constraint to mimic the local behavior of a traditional sequential storage and achieve causal consistency globally. Similarly, applications using the *Ancestor* begin constraint and the *Snapshot Isolation* end constraint would always see their own writes and maintain snapshot isolation within a branch. Alternatively, the *K-Branching* constraint explicitly bounds the degree of branching in the system, giving developers the ability to balance the performance benefit of allowing local branching with the degree of divergence that this entails.

By default, TARDiS uses the *Ancestor* begin constraint and *Serializability* end constraint. Though some applications may benefit from multiple consistency levels [47], in most cases this default will let programmers write almost unmodified code, without explicitly specifying constraints.

5.2 Coding with TARDiS

TARDiS' greater fidelity in capturing the context that leads to conflicting operations is key to reducing the complexity of developing ALPS applications and improving their performance. This feature is evident even in simple programs, such as the counter presented in Figure 3. In a traditional, non-branching causally consistent system, counters are often implemented as two separate vector clocks (one for increment operations, the other for decrement operations) with an entry for each replica [42]. Reading the value of a counter requires adding the values in the increment vector and substracting those in the decrement vector. Similarly, applying a remote operation requires merging the local increment and decrement vectors with those of the incoming remote operation by taking the maximum of each corresponding vector element. Thus, all operations, including non-conflicting reads, in effect involve a merge: the system must reconstruct the global view from each replica's local view, at a cost linear in the number of replicas.

```
1  func buy(customer, item, cart)
2    Tx t = client.begin(AncestorConstraint)
3    list<itemId> items = t.get(cart.items)
4    items += item.itemId
5    t.put(cart.items, items)
6    int stock = t.get(item.stock)
7    t.put(item.stock, stock-1)
8    list<cartId> carts = t.get(item.carts)
9    carts += cartId
10   t.put(item.carts, carts)
11   t.commit(SerializabilityConstraint)

13 func merge()
14   Tx t = client.beginMerge(AnyConstraint)
15   list<item> conflictItems =
16           t.findConflictWrites(t.parents)
17   forkPoint forkP =
18           t.findForkPoints(t.parents).first
19   foreach item in conflictItems
20     list<int> stockVals = new list
21     int forkPtStock = t.getForID(item.stock,forkP)
22     foreach branch in t.parents:
23       stockVals.add(t.getForID(item.stock,branch))
24       int newStock = mergeCounter(stockVals,
             forkPtStock)
25     if(newStock > 0)
26       t.put(item.stock,newStock)
27       confirmItem(item.itemId, item.carts)
28     else
29       // get orders since fork point
30       set<cartId> carts = new set
31       foreach branch in t.parents:
32         carts+= t.get(item.carts)
33       carts = carts - t.getForID(item.carts,forkP)
34       carts.sortBy(valueOfCart)
35       foreach cart in carts
36         if (forkPtStock > 0)
37           // confirm item until run out
38           --forkPtStock
39           confirmItem(item.itemId, cart.cartId)
40         else
41           // apologize to other users
42           removeRelatedItems(item,cart)
43           sendApology(cart.clientId)
44       t.put(item.stock,0)
45   t.commit(SerializabilityConstraint)
```

Figure 4: TARDiS' shopping cart implementation

In TARDiS, instead, single mode and merge mode are cleanly separated. In single mode (see Figure 3), increment and decrement operations access a single field, just as they would in a non-distributed scenario. Access to fork points makes merge operations both simpler and more flexible. In TARDiS, merging distinct counter branches is easy: one can simply compute the merged value by summing, for all branches, the difference between the value of the counter at the fork point and the current value for the branch. The application can then choose to merge branches periodically, during periods of low load, or to do so more frequently, if the counter nears boundary values.

The benefits of the TARDiS API become especially notable in examples that involve multiple objects with richer semantics. Consider the case of an online game store that sells both board games and extension packs that are only playable after buying the corresponding board game. The store tracks inventory by keeping a counter object per each item it sells, and associates each customer with a shopping cart. Suppose Alice and Bruno have, on different sites, both bought the last copy of a boardgame. Bruno has additionally bought an extension pack. Figure 4 gives the simplified pseudocode of the merging process.[1] On a merge, the application iterates over the keys in conflict and detects which items have been

[1]For clarity of explanation, we assume two branches only.

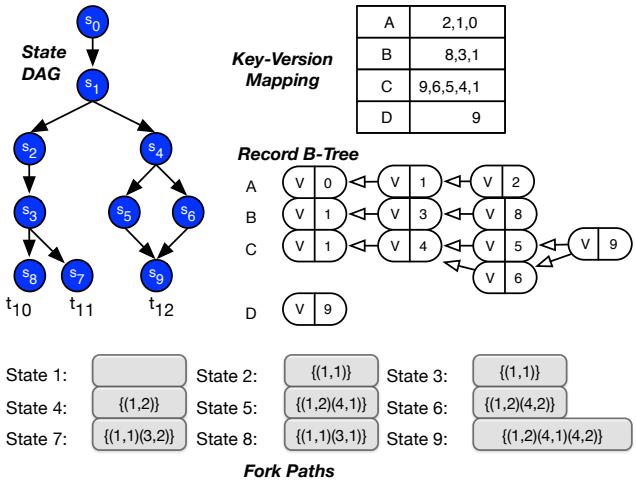


Figure 5: Main system datastructures

bought on different sites (lines 19-44), reconciling counter values through the merging process discussed above (lines 22-24). When the counter for a particular item falls below zero, as in our scenario, the merging logic must select the shopping cart from which the oversold items should be removed, while maintaining the invariant that no user should buy the extension pack without the game. The application has several options: it can choose, as does the pseudocode in Figure 4 (lines 30-44), to leave Bruno with both the game and the expansion pack, and send an apology to Alice, maximizing its overall profit. Alternatively, it can observe that Alice is a better customer than Bruno, and choose to privilege customer loyalty.

In current systems, merging that spans conflicts across multiple objects and requires application involvement is not achievable without significant engineering effort. Through the combination of branch-on-conflict, inter-branch isolation, and application-specific merge, TARDiS makes it easy for applications to acquire the context that led to such conflicts and empowers them with the flexibility and expressiveness necessary to meaningfully reconcile them.

6. DESIGN AND IMPLEMENTATION

To ensure that branches are cheaply created, maintained, and merged, TARDiS proceeds as follows. A transaction starts by identifying a most recent state that satisfies its begin constraint: this is the state from which the transaction can read (its *read state*). Likewise, upon commit, the transaction identifies a most recent state since its read state that satisfies the end constraint (the *commit state*). Since this process is identical for all transactions and independent of concurrently executing transactions, it naturally leads to state forking and to transactions aborting. If two concurrent transactions select the same state from which to commit, a new branch is created; alternatively, if no state satisfies a transaction's end constraint, the transaction aborts. There is thus no conceptual difference between sequential execution and forking, and TARDiS' design ensures that the implementation is similarly uniform. We describe this process in further detail below, focusing on the life of a particular transaction and relying on Figure 5 to illustrate the TARDiS' main datastructures.

6.1 Basic Operation

6.1.1 *Begin Transaction: Read State Selection*

A TARDiS transaction begins by selecting a branch. To choose one among the most recent suitable states, the transaction conducts

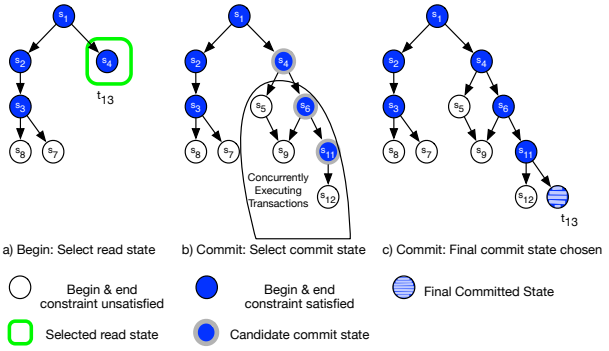


Figure 6: Transaction commit logic

a breadth-first search through the State DAG from its leaves up, looking for a state that satisfies the transaction's begin constraint. For example, given the State DAG in Figure 6(a), a newly executing transaction (t_{13}) would visit, in order, s_8, s_7, and s_4 and select the latter—the first state to satisfy the transaction's begin constraint—as its read state. Some constraints may require states to store additional information beyond pointers to their parents or children: the *Parent* and *StateID* constraints, for example, require all states in the DAG to be uniquely identifiable. Similarly, the *Serializability* and *Snapshot Isolation* (end) constraints demand to store, with each state, the read and write sets of the transaction responsible for creating it. In practice, there is often a unique state that satisfies the begin constraint; if instead multiple states are suitable, TARDiS simply selects one of them at random.

6.1.2 *Commit Transaction: Commit State Selection*

The process for committing a transaction is similar: it requires identifying the most recent state, since the read state, that satisfies the end constraint. At commit, the transaction first checks whether its read state satisfies the end constraint. If it does not, the transaction aborts, as it read from an invalid state. Otherwise, the transaction checks whether more recent states also satisfy the end constraint. In effect, starting from its read state, the transaction "ripples" down the DAG, stopping before the first state that no longer satisfies the end constraint. Figure 6(b) illustrates the process. Transaction t_{13} first checks that the read state s_4 satisfies the end constraint, and then ripples down through states s_6 and s_{11} until it encounters s_{12}. As s_{12} does not satisfy t_{13}'s end constraint, t_{13} commits after s_{11}, creating a new branch (Figure 6(c)).

6.1.3 *Reading records*

Logically, non-read-only transactions create a new database state every time they execute. Storing each of these states as physically distinct instances is unsustainable. Write operations therefore simply create new record versions, and TARDiS relies on system logic to reconstruct the appropriate state for reads. This copy-on-write approach is similar to that of most multiversioned concurrency control (MVCC) systems [12], but with one key difference: TARDiS must not only provide support for stale snapshots, but also for divergent snapshots. As in traditional MVCC systems, to read a record a transaction must determine the most recent record version in the key-version map. Unlike MVCC systems, however, TARDiS must also ensure that the record version belongs to the branch it has selected. The selected version is then read from the record B-tree and returned.

To quickly determine whether a record version belongs to the selected branch, TARDiS abandons traditional dependency checking [21, 31, 34], which quickly becomes a bottleneck in causally

```
1  descendantCheck(x, y):
2    if x.id = y.id then return true
3    else if x.id > y.id then return false
4    else if x.path ⊈ y.path then return false
5    else return true
```

Figure 7: Check if state y can see records associated with state x

consistent systems [8, 21], and instead relies on *fork point checking*. In TARDiS, a branch is summarized only by its fork points (§3). A fork point is a tuple (i, b), indicating that the current state is a descendant of the b^{th} child of state i. Together, the set of fork points for a branch denotes its *fork path*. A record version belongs to the selected branch if the fork path of the state associated with this record version is a subset of the fork path of the transaction's read state (see pseudocode in Figure 7). Figure 5 shows the fork path associated with each state: one can quickly determine that s_7 is on the same branch as s_3, as the fork path of s_3 is a subset of that of s_7. Similarly, s_9 is on the same branch as both s_5 and s_6.

By capturing *conflicts* (fork points) instead of dependencies, fork paths allow TARDiS to track concurrent branches efficiently. The small size of fork paths (conflicts are a small percentage of the total number of operations) not only limits memory overhead, but makes it possible to check quickly whether two states are on the same branch.

To guarantee that transactions select the most recent version, TARDiS keeps a topological sort of versions in each key-version mapping entry. Consider, for example, key C in Figure 5. The list stored in the key-version mapping is a topological order of the true structure of C's record versions, as saved in the record B-tree. As transactions iterate through the list, the first record version identified as belonging to the selected branch will necessarily be that branch's most recent version.

Putting this together, consider Figure 5. It shows three transactions t_{10}, t_{11}, and t_{12}, with respective read states s_8, s_7, and s_9. t_{10} would read *v2* for key A, *v8* for B, *v1* for C, and *empty* for D. Similarly, t_{11} would read *v2*, *v3*, *v1*, and *empty* respectively for keys A to D. Finally, t_{12} would read *v1*, *v1*, *v9*, and *v9*.

6.1.4 Writing values

To handle uniformly branching and non-branching scenarios, TARDiS's write logic ensures (*i*) that writes, whether conflicting or not, never block, and (*ii*) that updating the appropriate record on the correct branch is cheap.

Both aims can be very simply achieved by pushing most of the work to reads. As long as writes preserve the topological order of versions in the key-version mapping, the read logic ensures that the appropriate version is returned. Hence, a write operation in TARDiS simply creates a new record version storing the transaction's state identifier and the pertinent data, inserts it into the record B-tree, and appropriately updates the corresponding key-version mapping. Since state identifiers (and thus record identifiers) are monotonically increasing along a branch, TARDiS can cheaply maintain a topological order as a sorted list (more precisely, as a lock-free skip list). Thus, independently of whether conflicting writes occurred, all a transaction needs to do to complete a write is to insert the new version into the skip list.

Read-only Transactions Since read-only transactions cannot induce conflicts, TARDiS does not add them to the State DAG. This optimization limits unnecessary DAG growth, easing pressure on the garbage collector.

6.2 Merge Transactions

Merge transactions in TARDiS function similarly to single mode, but with a key difference: they select multiple read states, and hence operate on multiple branches. In merge mode, the application is thus directly exposed to any conflicting writes that forked the state of the datastore.

Merge transactions must atomically reconcile all conflicting objects, writing back a single merged state. As stated previously (§3), merging often requires providing the application with detailed information about the structure of the State DAG, including how branches diverged and the values of conflicting objects at the branches' fork points. This information must be made available efficiently, as a slow merge will stop applications from seeing up-to-date values. TARDiS thus provides an API to aid applications understand branch divergence. It consists of three operations: `findForkPoints`, to identify the fork point(s) of a set of branches; `findConflictWrites`, to list conflicting keys across branches; and `getForID`, to obtain, at the specified state, the record corresponding to a given key. TARDiS leverages the properties of the existing storage and consistency layers to make these operations fast. It implements `findForkPoints` by identifying the fork point(s) of the merge transaction's read states. For `findConflictWrites`, TARDiS similarly identifies the fork points of the branches and computes the conflicting key list from the write set of each intermediate state. Finally, for `getForID`, it uses the key-version mapping to select the appropriate record for a given state.

6.3 Garbage Collection

Most traditional databases store only the active frontier of records, keeping space overhead manageable as old transactions commit. TARDiS, on the contrary, stores by default all stale and parallel versions or states. For performance and efficiency under finite storage, TARDiS implements an aggressive, three-pronged garbage collection policy that runs concurrently with regular operations: (*i*) users place *ceilings* on specific states, promising never to use any state that precedes them as a read state; (*ii*) a path-compression algorithm compresses the State DAG to remove all states that are neither fork points nor leaf states; and (*iii*) a record-promotion algorithm removes record versions that are no longer visible because of ceilings or path compression.

Path Compression Path compression relies on one core heuristic: since most merging policies only require the fork points and the leaf states of a given execution, all intermediate states can be safely removed from the State DAG. In effect, this reduces the DAG to explicitly tracking the *nearest conflict dependency* using a three-staged process (Figure 8).

First, a *ceiling marking* bottom-up pass marks all the states above a recently placed ceiling. Marked states can no longer be selected as read states, ensuring that no new transaction starts above a ceiling (Figure 8(b)). Second, a *safe to garbage-collect* top-down pass labels as *safe-to-gc* all marked states (*i*) that are not currently selected as read states by some executing transaction and (*ii*) whose ancestors are also safe to garbage collect. This pass prevents committing transactions from rippling down deleted states and ensures that a state will only be deleted after all its ancestors have been deleted (Figure 8(c)). Finally, a *garbage collecting pass* marks safe-to-gc states that are not fork points as *garbage collectable* (*gc-able*). Garbage-collectable states are then "promoted" by mapping their state identifier to that of their most recent non-deleted child. All accesses looking up a deleted state are henceforth forwarded to the promoted state. In effect, the child node takes over the identity of its parent as well as its own, allowing for the parent to be garbage collected (Figure 8(d.e.f)). Consider in Figure 8 an object D that is last modified in s_1. Its state identifier is therefore 1. All transactions whose read states are a descendant of s_1 see D. But, when s_1

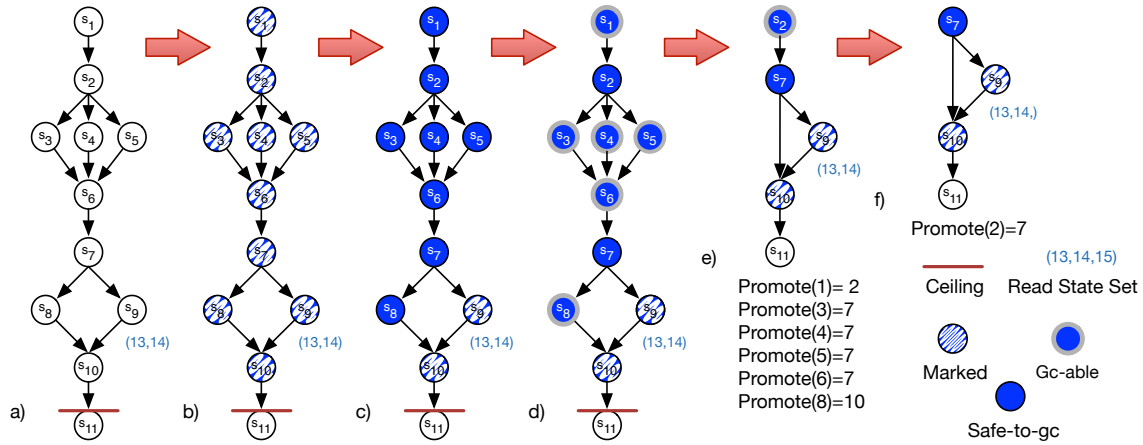


Figure 8: Path Compression Algorithm - Ceiling placed above s_{11}. s_8 and ancestors are marked as safe-to-gc; since s_{10} is the read state for several pending transactions, it cannot be marked as safe-to-gc. Non fork points safe-to-gc states are marked as gc-able and deleted.

is garbage collected, any transaction that tries to read D and thus looks up the fork path of s_1 would fail. Promoting s_1 to s_7 ensures that any such transaction is redirected to s_7. Once promoted, garbage collectable states can safely be removed.

Record Promotion Next, TARDiS deletes record versions that are no longer needed. Record versions associated with previously deleted states are promoted to their first non-garbage-collected descendants and updated to reflect their new identifier. This promotion creates long chains of records with the same state identifier. Since TARDiS' get/put algorithm returns only the most recent visible record, none but the first of the promoted records that share an identifier will ever be accessed: the rest can therefore be safely discarded. A full record-promotion pass, as in path compression, hence ensures that the only record versions maintained are those that are either current or at a fork point.

6.4 Replication

The Replicator (§4) uses a gossip protocol [1] to asynchronously propagate locally executing transactions. These transactions carry a *StateID* constraint that specifies the state to which they should be applied. The Replicator applies a newly received transaction if the required parent state is present. If not, the transaction is cached to be appended later. The *StateID* constraint removes the need to track expensive dependency meta-data: it reduces dependency checking to looking up whether the state with the corresponding id is present in the remote DAG, which can be done in constant time.

Garbage collection is triggered by each local Replicator and can operate either optimistically or pessimistically. When pessimistic, TARDiS garbage collects states only after receiving unanimous consent from all Replicators [34]. As this can cause garbage collection to trail significantly during partitions, optimistic mode lets sites garbage collect states independently. If a replica later needs a state it garbage-collected, the replica simply retrieves the missing information from the appropriate replica. This may cause some transactions to block. If an application erroneously places a ceiling that causes states to be prematurely garbage collected, TARDiS simply aborts the transactions that try accessing the missing states.

The current prototype of TARDiS does not support data partitioning, but it can be extended to support this feature by using an approach similar to the one taken by COPS [31]—in essence, by executing distributed transactions within a datacenter (with the State DAG collocated with the transaction manager) and replicating transactions asynchronously across datacenters.

6.5 Fault Tolerance and Recovery

TARDiS guarantees consistency, atomicity, and (optionally) durability if, at all times, the write operations of all transactions present in the State DAG are observable by future transactions. TARDiS maintains this invariant across failures by logging, at transaction commit time, the id of the corresponding commit state, its parent state(s) ids, and the transaction's write set keys.

Recovery During recovery, TARDiS reconstructs the State DAG and key-version mapping by iterating chronologically over the log. For each log entry, the recovery process (*i*) inserts a new state in the DAG with matching id and adds it as a child of the states with matching parent ids; and (*ii*) adds a new entry in the key-version mapping for each key in the recorded write set. Iterating over the log chronologically guarantees that no child is recovered before its parents; the skip-list implementation of the key-version mapping guarantees that the order of entries in the key-version mapping is preserved across failures.

Asynchrous Flush To mitigate the overheads of writing to disk, TARDiS offers the option of asynchronously flushing both record updates and the commit log (at the cost of durability). To preserve atomicity, TARDiS ensures that the commit log is flushed sequentially and further checks, on recovery, whether each entry in the write set has been made persistent to stable storage. If only part of a transaction's effect has been made persistent, the corresponding state and all subsequent states are discarded. Orphaned records (resulting from operations that belong to, or depend on, partially applied transactions) have no effect on correctness, as it is the DAG and key-version mapping that determine what can be read. These records are simply eventually garbage collected.

Checkpointing To reduce log size, TARDiS periodically takes non-blocking checkpoints by (*i*) selecting a state id s_c, (*ii*) flushing all outstanding writes, and (*iii*) saving every DAG state with id smaller than s_c.

Replication To recover transactions that have committed elsewhere but are lost locally, the recovery process broadcasts a vector with the id of the latest surving state received from each Replicator. Replicas respond by sending states and records more recent than their corresponding entry in the vector. If these states have already been garbage collected, it may be necessary to send the full DAG.

6.6 Implementation notes

Our prototype consists of 15K lines of Java in two configurations. In TARDiS-BDB, record persistence to disk is via Berke-

leyDB with concurrency control turned off, while TARDiS-MDB depends on MapDB [35]. We rely on Google Protobuffers v2.4.5 for message serialization and on the Netty networking library for inter-site communication.

7. EVALUATION

TARDiS proposes branching as the fundamental abstraction to model conflicts end-to-end. To quantify the cost and benefits of this new abstraction with respect to both performance and complexity, we first benchmark the performance of the system itself. We then use the abstraction to program two representative ALPS applications: CRDTs [42], a library of scalable, weakly consistent datatypes (counters, sets, maps, and more); and Retwis [41], a Twitter clone. We find that using state branching/merging cuts the number of lines of code in half, while judicious branch-on-conflict yields a speedup of between two and eight times.

7.1 Microbenchmarks

Our microbenchmarks answer the following questions:

(i) What are the overheads of tracking state? (§7.1.2,§7.1.5)
(ii) When does local branching improve performance? (§7.1.3)
(iii) What impact has expressiveness on performance? (§7.1.4)
(iv) How does the system scale for multiple sites? (§7.1.6)

7.1.1 Evaluation Setup

Unless otherwise stated, we run our experiments on a shared local cluster of machines equipped with a 2.67GHz Intel Xeon CPU X5650 with 48GB of memory and connected by a 2Gbps network. Inter-machine ping latencies average 0.15 ms. Each experiment is run with three dedicated server machines, three dedicated Replicators and with clients spread equally among separate machines. To the best of our ability, all runs were performed in the absence of interfering workloads.

We compare TARDiS with both BerkeleyDB v6.2.31 Java Edition (**BDB**) and a custom OCC implementation (**OCC**) that uses BDB as its backend. BerkeleyDB, as a widely-used, pure-Java ACID datastore, provides a sound basis for comparison against our TARDiS prototype. We configure BerkeleyDB so that (i) read-write transactions are flushed to disk asynchronously; and (ii) all requests hit the cache. Our OCC implementation is based on a modified version of Kung et al.'s algorithm [28] that does not require read-write transactions to be verified against read-only transactions.

Each client issues transactions consisting of six operations in a closed loop, running for two minutes. We use a set of clients large enough to saturate the system. Read-only transactions contain only reads; read-write transactions contain three reads and three writes. We consider four transaction mixes differentiated by the ratio of read-only transactions to read-write transactions: Read-only (R-O, 100/0), Read-heavy (R-H, 75/25), Mixed (M, 25/75), and Write-Heavy (W-H, 0/100). We report only on the read-heavy and write-heavy workloads. We further consider two access patterns from the YCSB benchmark [18]: uniform and Zipfian (p=0.99). We report on TARDiS-BDB only: the performance of TARDiS-MDB is similar (approximately 10% better).

7.1.2 Baseline TARDiS performance

We begin by establishing a baseline for the performance of TARDiS: we want to determine how it compares, when local branch-on-conflict is not enabled, against both our simple implementation of OCC and a commercial system like BDB. Transactions use *Ancestor* as begin constraint and the union of *Serializability* and *No Branching* as end constraint. We expect this setup to be common: it guarantees that each application sees its own writes and that the execution is serializable.

Workload		Begin	Get	Put	Commit
RH-Uniform	TARDiS	0.6	0.6	1	0.2
	BDB	0.3	0.4	1.2	0.1
	OCC	0.5	0.6	1.1	3.5
WH-Uniform	TARDiS	1	1.2	1.1	1.8
	BDB	0.6	0.8	2.7	0.1
	OCC	0.9	1.2	1.1	6.7
WH-Zipfian	TARDiS	1	1.4	1.2	0.8
	BDB	0.6	8	23	0.1
	OCC	0.4	0.9	1.2	9

Table 3: Per-operation latency breakdown ($\times 10^{-2} ms$)

The throughput-latency graph in Figure 9 shows that, despite tracking the full system's history and paying the overhead of selecting a read and commit state, TARDiS-BDB performs similarly to BDB for both read-heavy and write-heavy workloads. TARDiS incurs a 10% slowdown as its begin and commit phase are more costly than in BDB. As contention increases, however, TARDiS-BDB's lock-free writes reduce the performance gap. In both cases, our OCC implementation lags behind. For the read-heavy workload, OCC must verify read-only transactions, which increases latency and reduces throughput. For the write-heavy workload, OCC suffers from a long validation phase. Though also optimistic, TARDiS's validation phase (commit state identification) is less costly: it requires checking only the write set of those transactions that already committed as children of the selected read state. In contrast, OCC requires checking against all concurrently committing transactions.

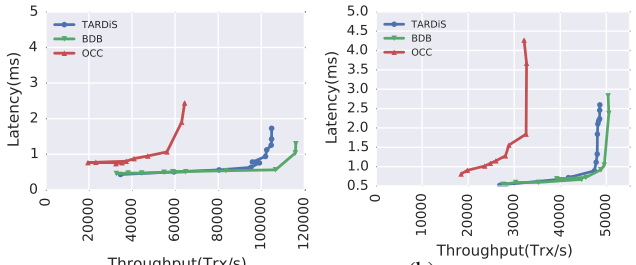
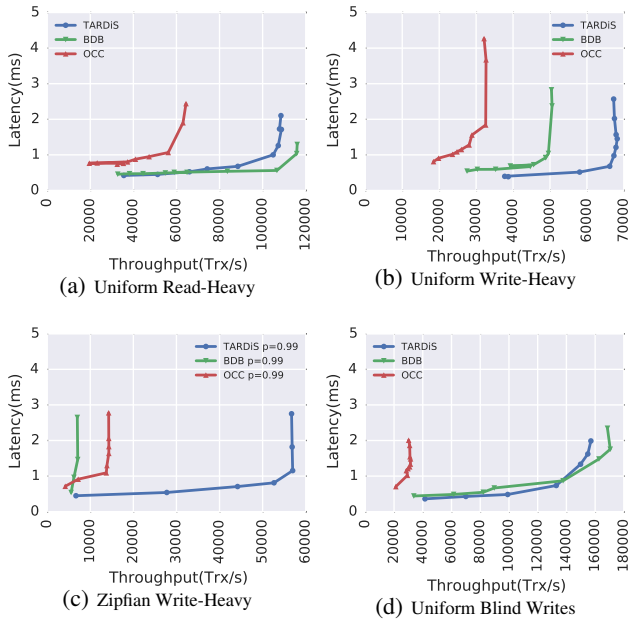


Figure 9: TARDiS-BDB vs BerkeleyDB vs OCC

7.1.3 Impact of branching

TARDiS lets ALPS applications choose to branch on conflict rather than abort. Unsurprisingly, we find that the relative benefits of branching increase with contention. Results are shown in Figure 10 (all transactions run with branch-on-conflict enabled and with *Ancestor* and *Serializability* as, respectively, their begin and end constraint). Table 3 provides a per-operation breakdown of the same experiments, excluding network latency and retries. Figure 10(a) shows that, when contention is low, branching indeed does not help: TARDiS's performance is slightly lower than BDB's. By contrast, with higher contention (Figure 10(b)), TARDiS outperforms BDB by 35%. The performance of BDB drops by half, as transactions wait longer for locks to become available, increasing the cost of gets and puts by a factor of almost two (Table 3 shows that reads take 0.004ms in the R-H workload, and 0.008ms in the W-H workload). The 37% throughput drop in TARDiS is due to several factors. First, the lack of read-only transactions: since all transactions now have to identify a commit state, the commit cost of the transaction increases (from Table 3: from 0.002ms to 0.018ms). Second, reads become more expensive, as more record versions



(a) Uniform Read-Heavy (b) Uniform Write-Heavy

(c) Zipfian Write-Heavy (d) Uniform Blind Writes

Figure 10: Benefit of branching as a function of workload

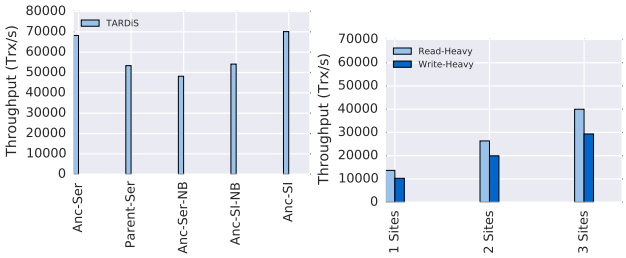
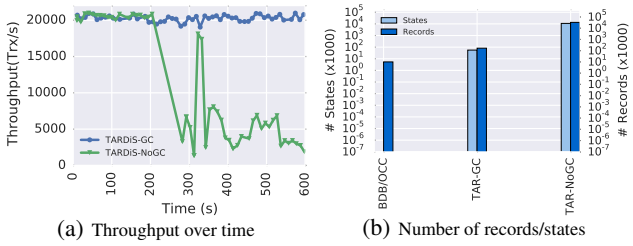


Figure 11: Constraint Choice *Figure 12: TARDiS Scalability*

need to be checked (from 0.006ms to 0.012ms). With both low and high contention, the throughput of OCC is bottlenecked by the verification phase. The relative performance increase of branching over sequential storage is most marked when requests follow a Zipfian workload in which a small number of objects, accessed very frequently, experience a high degree of contention (Figure 10(c)). In this scenario, TARDiS outperforms BDB by a factor of eight. Whereas moving from a uniform to a Zipfian distribution causes BDB's performance to collapse (7x throughput decrease), TARDiS' throughput is much less affected: the lock-free implementation of the write skip-list ensures that writes never block, even when conflicting. Table 3 confirms this: the cost of writes increases only moderately in TARDiS (from 0.011ms to 0.012ms). Similarly, the use of fork points to efficiently summarize the DAG means that, despite an eight-fold increase in the branching factor in the Zipfian workload, the cost of reads only increases by 16%. In contrast, locking causes the read and write time for BDB to increase by a factor of ten. OCC performs comparatively better, as it ensures that (*i*) at least one transaction will always commit, and (*ii*) readers will not block writers. Nonetheless, its high abort rate and expensive validation phase limit its throughput to a fifth of TARDiS'.

Branching, however, is not always beneficial. In the workload of Figure 10(d) transactions consist of a single write accessing the database uniformly, conflicts are rare, and locks are held for a very short period of time: here, branching does not help, but still incurs the cost of tracking past states. Since the current implementation of TARDiS' garbage collector is unable to keep up with the speed



(a) Throughput over time (b) Number of records/states

Figure 13: Impact of garbage collection

at which new states are generated, the increased memory pressure grows the number of stalls induced by garbage collection, causing TARDiS to perform 10% worse than BDB.

7.1.4 Impact of Constraint Choice

The choice of begin and end constraints involves a complex trade-off between consistency and performance. To shed some light on this trade-off for our current implementation, we plot in Figure 11 the throughput of TARDiS for several different constraint choices for the same configuration (15 machines, each with seven clients) that led TARDiS to reach the elbow in the write-heavy throughput/latency graphs of Figures 9 and 10. We focus on the *Ancestor* and *Parent* begin constraints because we expect them to be the most popular. *Ancestor* ensures that clients will see their own writes, along with those of any non-conflicting clients. In the Wikipedia example, this would allow Alice to read back Carlo's operations, but not Davide's or Bruno's. *Parent* results in a behavior very much akin to that of a local Git branch, as clients will see their own operations only. A detailed per-operation performance breakdown (omitted for lack of space) gives us clues for why, despite commit selection being 30% cheaper in *Parent* (as only the read state can satisfy it), *Ancestor* still outperforms *Parent* by 21%. First, read state selection is 40% more expensive in *Parent*, as it requires a look-up over the full DAG rather than one that only involves its leaves. Second, since *Parent* results in more branches, fork path checking becomes more expensive, increasing the cost of reads by 15%. Finally, *Parent* prevents states from being quickly garbage collected.

Unlike begin constraints, we find that end constraints, as long as they do not cause transactions to abort, do not significantly affect throughput or latency: throughput results for the (branching) *Serializability* and *Snapshot Isolation* constraints are within 5% of each other, mostly because the former, by creating twice as many branches, increases the cost of reads by 10%. Non-branching serializability and snapshot isolation both perform poorly in comparison. Though each individual operation is cheap, they see repeated aborts due to write-write or read-write conflicts.

7.1.5 Garbage Collection

Figures 13(a) and 13(b) report, respectively, the throughput and number of DAG states and records generated by a single server running TARDiS, with and without garbage collection, over a ten minute run with *Ancestor* begin constraint, *Serializability* end constraint, and clients placing ceilings (§6.3) every 1000 transactions. Without DAG compression, throughput drops dramatically after three minutes, since old and new generation Java garbage collection pause TARDiS constantly. With DAG compression, throughput instead remains constant, as (*i*) states are removed from the DAG and their underlying datastructures recycled for incoming transactions, and (*ii*) record promotion/deletion keeps the key-version mapping structure small. Compression should ideally bound the DAG size to the product of the number of clients times the ceiling placing

interval (i.e., $30 \times 1000 = 30,000$ states) but, as Figure 13(b) shows, the DAG is $55,000$—98% fewer states than without garbage collection, but still two times higher than the ideal. This mismatch is due to record compression. Though, in principle, states can be removed from the DAG as soon as they become garbage-collectible, in practice the promotion table must first be flushed, which can only happen after a full record promotion pass. Unfortunately, these records are often not present in the cache and must be read from disk, causing the record promotion threads to trail behind. Likewise, the storage size in TARDiS is 15x larger than in OCC or BDB. We are currently investigating alternative schemes that do not have this drawback.

7.1.6 Replication

We measure the scalability of TARDiS when running in geographically distinct replicas. We use a Google Cloud Services cluster of three machines (2.5GHz Intel Xeon E5 v2 machines with 60GB of memory), running in three geographical zones (us-central1-f, europe-west1-b, asia-east-1). Figure 12 shows how the aggregated throughput scales with the number of sites. As transactions are asynchronously replicated, latency is unchanged from the single site experiments. TARDiS scales linearly with the number of sites, as its design ensures that transactions, when applied to remote sites, do not contend with local transactions.

7.2 Applications

In porting ALPS applications to TARDiS, we answer two questions: (*i*) Do explicit state branching and merging simplify conflict resolution? (*ii*) Can ALPS applications use local branch-on-conflict to improve their performance?

7.2.1 Simplifying Merging: CRDTs

Convergent Replicated Data Types (CRDTs) [42] are a family of data types that support lazy replication by including a set of semantically meaningful conflict resolution functions. A variety of CRDTs have been developed [40, 42], including counters, sets, registers, and trees. To date, they have been incorporated into cooperative text editing tools [40] and in the Riak key-value store [26].

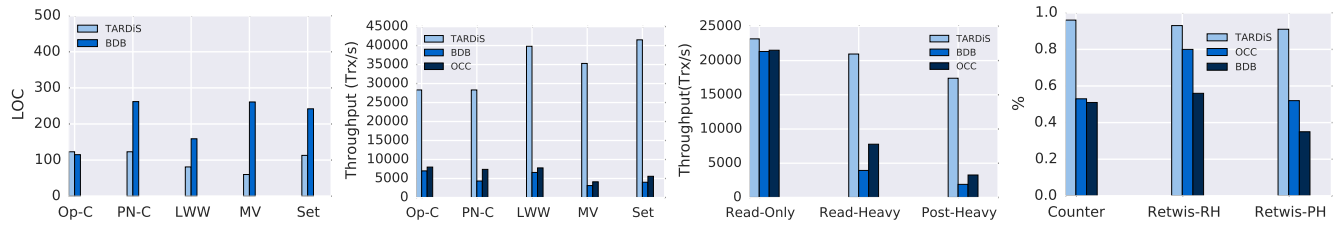
We followed the algorithms developed Shapiro et al. [42] to implement, on top of both TARDiS and BDB, a subset of CRDTs sufficient to design realistic applications. The TARDiS implementation proved easier to write and required less code, often by a factor of two or more (Figure 14(a)): the entire effort took less than a day with TARDiS, compared to over three days with BDB. Two features of TARDiS account for this: *StateID* replication and conflict tracking. *StateID* replication guarantees that operations that execute locally on a given state will execute on the same state at all remote sites, and therefore eliminates the need to capture and replicate side-effects. In turn, conflict tracking makes it easy, for each CRDT replica implemented in TARDiS, to access, through the State DAG, a consistent view of the updates that need to be merged. Implementations on flat storage systems like BDB, in contrast, need to explicitly track the updates applied at each replica and make sure that the state of the CRDTs is replicated consistently everywhere. Consider, for example, the counter CRDT, which is modeled as two separate increment and decrement vectors, containing an entry per replica. On BDB, it is up to the CRDT developer to ensure that, as new operations are applied, the global state is tracked correctly at each replica. TARDiS instead tracks the necessary information by design. With access to the fork point, merging becomes as easy as adding, for each branch, the difference between the counter's value at the fork point and at the current state.

To guarantee eventual convergence, CRDTs implemented on sequential storage must mutate local state atomically and sequentially (e.g., new vector clocks must be created atomically to guarantee causal delivery, and updating counters requires read-modify-write operations). Each replica must consequently be serializable, thus limiting the throughput at each site. Branch-on-conflict removes this limitation without sacrificing consistency. For a workload consisting of 90% reads and 10% writes, with transactions configured to use the *Ancestor* and *Serializability* constraint set and periodic merging, TARDiS' CRDT implementations achieve a four to eight times speedup over their sequential counterparts (Figure 14(b)). Three factors contribute to this speedup. First, each individual operation is simpler: for instance, reading or updating a counter no longer involves manipulating a vector, but simply requires reading or writing an integer. Second, operations are no longer serialized, but fork on conflict, and are later merged back. Finally, merges can be batched. Traditional CRDTs require a merge for every remote operation received; in TARDiS, merges need only take place periodically, as operations are consistently recorded in separate branches. These effects are displayed in Figure 14(d). It shows, for a CRDT counter implemented in TARDiS, BDB, and OCC, the percentage of *useful work*, measured as the time spent executing committed transactions (i.e., excluding time spent waiting on locks, aborted transactions, and merge transactions): useful work in TARDiS is at 0.96, while for BDB and OCC almost half the time is wasted.

7.2.2 Pushing weak consistency down: Retwis

To understand the performance implications of building ALPS applications in TARDiS, we implemented Retwis [41], a popular Twitter clone [39, 44, 51]. Retwis users can create accounts (*createAccount*), follow users (*followUser*), post new content (*post*), and read their own timeline (*readOwnTimeline*), which includes their own tweets and those of the users they follow. In our implementation, *readOwnTimeline* returns the 50 most recent posts. Contention primarily arises when a user posts new content, as the Retwis implementation must ensure that the tweet becomes visible to all the user's followers. Retwis, like many ALPS applications, has relatively weak consistency requirements: as long as posts are not incorrectly attributed and are presented in causal order, users can tolerate small delays in seeing posts. Porting Retwis on TARDiS was straightforward. We simply extended each transactional call to take the *Ancestor* begin constraint and *Serializability* end constraint, and implemented a separate conflict resolver that periodically merges conflicting branches by resolving duplicate user ids and merging timelines (preserving the order of posts).

Figure 14(c) shows the throughput of Retwis on TARDiS, BDB, and OCC, for three workloads: read-only (100% reads), read-heavy (85% reads, 5% follows, and 10% posts), and post-heavy (65% reads, 5% follows, and 30% posts). Branching does not benefit the read-only workload, but it significantly softens the performance blow caused by contention in the remaining two workloads. An analysis of the per-transaction behavior (omitted for lack of space) reveals that the throughput of *readOwnTimeline* operations drops by 70% for OCC in the read-heavy workload, as *posts* cause these transactions to abort, and by 80% in BDB, as write operations block both reads and other writes (causing in turn read operations to block for longer). These effects are amplified in the post-heavy workload. Moving to Figure 14(d), we see that TARDiS, by branching and merging asychronously, is able to maintain a much higher fraction of useful work than OCC and BDB for both of these worklods because, unlike waiting on locks, merging does not prevent the system from making progress. The small drop in TARDiS' throughput is due to the need to identify commit states for *posts*. Thus, for ALPS



(a) CRDT Lines of Code (b) CRDT Throughput (c) Retwis Throughput (d) Application Goodput

Figure 14: Applications on BerkeleyDB and TARDiS. Op-C:Operation Based Counter, PN-C: State Based Counter, LWW: Last-Writer-Wins Register, MV: Multivalued Register, Set: Or-Set

applications, executing conflicting transactions optimistically and reconciling them later can improve scalability within a site, much like weak consistency improves scalability across sites.

8. RELATED WORK

Avoiding Conflicts The core challenge in geo-replication is handling conflicts at different sites. One option is to preemptively ensure that conflicts do not happen, either through strong synchronization [2, 13, 19] or through scheduling transactions to avoid conflicts (using static analysis [53] or upon admission [49]). TARDiS explicitly targets applications whose availability or latency constraints preclude this option.

Weakening Consistency Systems designed for ALPS applications trade-off strong consistency for performance and provide weaker guarantees such as causal consistency [4, 20], timeline consistency [17], parallel snapshot isolation [44], and non-monotonic snapshot isolation [7]. Others have sought to give applications the flexibility of adapting the consistency level required as a function of the operation (Pileus [47], Red-Blue Consistency [29], MDCC [27]) or the object (Continuous Consistency Model [52], CRDTs [42], Escrow Transactions [43]). TARDiS instead allows for general conflict resolution strategies defined by the application.

Resolving Conflicts When conflicts are allowed, most systems resolve conflicting executions by projecting them onto sequential storage. COPS [31] adopts a first-writer-wins policy; Ficus [25], Dynamo [20], and Bayou all leave resolution up to the users; and operational transforms [45] leverage specific textual properties. Unlike in TARDiS, these resolution functions are per-object and do not allow programmers to see and resolve the entire state atomically.

Branching Conflicts in a distributed system introduce implicit branching that must be reconciled when conflicts are detected. A number of systems expose this branching: version control systems (Git) allow users to operate on different branches; the Olive [3] file-system allows users to create/and fork snapshots efficiently; and ORI [36] tracks possibly divergent histories across multiple devices. They contrast with TARDiS as their branching is explicit (git branch) rather than implicit: explicit branching requires synchronization, which is precisely what ALPS applications want to avoid. Some causally consistent systems also allow for concurrent writes to be exposed to the users (Ficus [25], Dynamo [20]). This is a limited notion of branching, which forks individual objects rather than a state. These systems, as a result, provide neither conflict tracking nor branches, and increase complexity for the application by systematically exposing it to multiple values. By contrast, in TARDiS programmers only deal with multivalued objects if they explicitly request it in merge mode. In the Byzantine context, SUNDR [30] and FAUST [15] develop fork consistency/linearizability to isolate clients that see different values in a faulty server, and Depot [34] extends SUNDR's model to support fork-joining. Depot does not, however, expose the abstraction of branches, and provides no support for cross-object atomic merges, unlike TARDiS. Finally, Sporc [22] does provide atomic joining of forks, but only for the restricted case of collaborative text applications, and uses operational transformation to resolve conflicts.

9. CONCLUSIONS AND FUTURE WORK

This paper introduced TARDiS, a novel transactional key-value store designed to support weakly-consistent systems. By explicitly tracking concurrent branches and exposing them, when needed, to applications, TARDiS simplifies conflict resolution. By giving applications the option of applying weak consistency principles end-to-end, TARDiS can significantly improve the performance of the local site.

TARDiS nonetheless lacks several features. First and foremost, its replicas do not support distributed transactions: we are currently investigating how to add this feature to the system. More generally, we believe that TARDiS' ability to efficiently distinguish between concurrent threads of execution makes it a strong candidate for concurrency control systems based on speculation. We hope to investigate whether speculation can alleviate many of the throughput issues in wide-area strongly consistent systems.

10. ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers and Phil Bernstein, Ionel Gog, Steven Hand, Kim Keeton, Akbar Syed Mehdi, Reinhard Munz, Clement Pit-Claudel, Daniel Porto, Rodrigo Rodrigues, Marco Serafini, Chunzhi Su, Malte Schwarzkopf, Chao Xie, Michael Walfish, John Wilkes, and Emmett Witchell for their insightful comments and feedback on earlier drafts of this work. This work was supported by the National Science Foundation under grant number CNS-1409555 and by a Google Cloud Platform credit award. Natacha Crooks was partially supported by a Google European Fellowship in Distributed Computing. A significant portion of this work was done while Allen Clement and Natacha Crooks were at MPI-SWS.

11. REFERENCES

[1] Gossip-based computer networking. *ACM SIGOPS Operating Systems Review*, 41(5), 2007.

[2] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, SOSP '07, pages 159–174.

[3] M. K. Aguilera, S. Spence, and A. Veitch. Olive: Distributed point-in-time branching storage for real systems. In

Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation - Volume 3, NSDI '06.

[4] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. Hutto. Causal memory: Definitions, implementation and programming. Technical report, Georgia Institute of Technology, 1994.

[5] P. Alvaro, P. Bailis, N. Conway, and J. M. Hellerstein. Consistency without borders. In *Proceedings of the 4th ACM Symposium on Cloud Computing*, SOCC '13, pages 23:1–23:10.

[6] Apache. Cassandra. http://cassandra.apache.org/.

[7] M. S. Ardekani, P. Sutra, and M. Shapiro. Non-monotonic snapshot isolation: Scalable and strong consistency for geo-replicated transactional systems. In *Proceedings of the 32nd International Symposium on Reliable Distributed Systems*, SRDS '13, pages 163–172.

[8] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. The potential dangers of causal consistency and an explicit solution. In *Proceedings of the 3rd ACM Symposium on Cloud Computing*, SOCC '12, pages 22:1–22:7.

[9] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Bolt-on causal consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13.

[10] Basho. Riak. http://basho.com/products/.

[11] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *ACM SIGMOD Record*, volume 24, pages 1–10, 1995.

[12] P. A. Bernstein and N. Goodman. Multiversion concurrency control;theory and algorithms. *ACM Transactions on Database Systems*, 8(4):465–483, 1983.

[13] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. 1987.

[14] E. A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing*, PODC '00.

[15] C. Cachin, I. Keidar, and A. Shraer. Trusting the cloud. *SIGACT News*, 40(2):81–86, June 2009.

[16] Cassandra. Cassandra Use Cases. http://www.planetcassandra.org/apache-cassandra-use-cases/.

[17] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment*, 1(2):1277–1288, 2008.

[18] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154.

[19] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '12, pages 251–264.

[20] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of 21st ACM Symposium on Operating Systems Principles*, SOSP '07, pages 205–220.

[21] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel. Gentlerain: Cheap and scalable causal consistency with physical clocks. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 4:1–4:13.

[22] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. SPORC: Group collaboration using untrusted cloud resources. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '10.

[23] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[24] Git. Git: the fast version control system. http://git-scm.com.

[25] R. G. Guy, J. S. Heidemann, W. Mak, T. W. Page, Jr., G. J. Popek, and D. Rothmeir. Implementation of the Ficus Replicated File System. In *Proceedings of the Summer 1990 USENIX Conference*, pages 63–72, 1990.

[26] R. Klophaus. Riak core: Building distributed applications without shared state. In *ACM SIGPLAN Commercial Users of Functional Programming*, CUFP '10.

[27] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. Mdcc: multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 113–126.

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

[29] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '12, pages 265–278.

[30] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation - Volume 6*, OSDI '04.

[31] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416.

[32] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '13, pages 313–328.

[33] P. Mahajan, L. Alvisi, and M. Dahlin. Consistency, availability, convergence. Technical Report TR-11-22, Computer Science Department, University of Texas at Austin, May 2011.

[34] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. *ACM Transactions on Computer Systems*, 29(4):12, 2011.

[35] MapDB. MapDB: Embedded Database Engine. http://www.mapdb.org/.

[36] A. J. Mashtizadeh, A. Bittau, Y. F. Huang, and D. Mazières. Replication, History, and Grafting in the Ori File System. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP '13, pages 151–166.

[37] MongoDB. Agility, Performance, Scalibility. Pick three. https://www.mongodb.org/.

[38] M. A. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '99.

[39] R. Padilha and F. Pedone. Augustus: Scalable and robust storage for cloud applications. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 99–112.

[40] N. Preguica, J. M. Marques, M. Shapiro, and M. Letia. A commutative replicated data type for cooperative editing. In *Proceedings of the 29th IEEE International Conference on Distributed Computing Systems*, ICDCS '09, pages 395–403.

[41] Retwis. Twitter-like Clone. http://retwis.redis.io/.

[42] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Rapport de recherche RR-7506, INRIA, Jan. 2011.

[43] L. Shrira, H. Tian, and D. Terry. Exo-leasing: escrow synchronization for mobile clients of commodity storage servers. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '08, pages 42–61.

[44] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pages 385–400.

[45] C. Sun and C. Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work*, CSCW '98, pages 59–68.

[46] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of the 3rd International Conference on on Parallel and Distributed Information Systems*, PDIS '94, pages 140–150.

[47] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP '13, pages 309–324.

[48] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, SOSP '95, pages 172–182.

[49] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 1–12.

[50] Wikipedia. Wikipedia: Conflicting Sources. http://en.wikipedia.org/wiki/Wikipedia:Conflicting_sources.

[51] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha. Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP '13, pages 292–308.

[52] H. Yu and A. Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation - Volume 4*, OSDI '00.

[53] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. Transaction chains: Achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP '13, pages 276–291.