11-3-2016

Achieving Causal Consistency under Partial Replication for Geo-distributed Cloud Storage

Tariq Mahmood
Purdue University, tmahmood@purdue.edu

Shankaranarayanan Puzhavakath Narayanan
Purdue University, spuzhava@purdue.edu

Sanjay Rao
Purdue University, sanjay@purdue.edu

T. N. Vijaykumar
Purdue University, vijay@purdue.edu

Mithuna Thottethodi
Purdue University, mithuna@purdue.edu

Achieving Causal Consistency under Partial Replication for Geo-distributed Cloud Storage

Tariq Mahmood, Shankaranarayanan Puzhavakath Narayanan, Sanjay Rao, T. N. Vijaykumar, and Mithuna Thottethodi

Abstract—Causal consistency has emerged as an attractive middle-ground to architecting cloud storage systems, as it allows for high availability and low latency, while supporting stronger-than-eventual-consistency semantics. However, causally-consistent cloud storage systems have seen limited deployment in practice. A key factor is these systems employ full replication of all the data in all the data centers (*DCs*), incurring high cost. A simple extension of current causal systems to support partial replication by clustering *DCs* into *rings* incurs availability and latency problems. We propose *Karma*, the first system to enable causal consistency for partitioned data stores while achieving the cost advantages of partial replication without the availability and latency problems of the simple extension. Our evaluation with 64 servers emulating 8 geo-distributed *DCs* shows that *Karma* (i) incurs much lower cost than a fully-replicated causal store (obviously due to the lower replication factor); and (ii) offers higher availability and better performance than the above partial-replication extension at similar costs.

Index Terms—Causal Consistency, Partial Replication, Cloud Storage.

1 Introduction

Cloud storage is one of the pillars on which the entire cloud infrastructure rests. The application layers of the cloud rely on the storage tier to offer low-latency, reliable, available, consistent storage over geo-distributed scales [11], [13], [15], [23], [27]. However, these goals are often at odds with one another. In fact, the CAP theorem [20] (even the more nuanced reading [8]) rules out certain strong flavors of consistency (e.g., linearizability [13]) for wide-area systems that are available and partition-tolerant. At the other extreme, eventual consistency [15], [23] ensures liveness but offers no static guarantees of when a value may become visible. Barring niche applications (e.g., banking), many cloud applications are satisfied with weaker consistency models than linearizability – however, eventual consistency is inadequate in many scenarios including those requiring causal ordering of events.

Causal consistency [2], [16], [17], [25], [27], [28], has emerged as an attractive middle-ground for cloud storage systems since it preserves the intuitive happened-before relationship, critical in many scenarios (e.g., announcements of price drops reaching customers who then discover the old (undiscounted) prices).

Causally consistent storage systems, ensure that the global ordering of operations respects each thread's program order as well as the (transitive) ordering implied by any inter-thread value communication, while staying available and partition-tolerant.

- *At the time this work was done, all authors were with the Department of Electrical and Computer Engineering, Purdue University, West Lafayette, IN, 47907.*
  E-mails: (tmahmood, spuzhava, sanjay, vijay, mithuna)@purdue.edu
- *Shankaranarayanan Puzhavakath Narayanan is currently with AT&T Research.*
  E-mail: snarayanan@research.att.com

Despite these advantages, causally consistent systems have seen limited adoption in practice. A key factor is that current causally-consistent, distributed cloud storage systems [2], [16], [17], [27], [28], suffer from a key drawback that effectively renders them impractical; they require *full replication*, where all the data is replicated in all the data centers (*DCs*). Such full replication is infeasible because of the immense size of the data stores as well as the large numbers of *DCs*.

Partial replication, where each data object is replicated in a subset of *DCs*, has been employed to reduce costs in eventually-consistent (e.g., [15], [23], [38]) or linearizable systems (e.g., [13]). Extending causal systems to support partial replication is however not easy. Current causal systems [16], [27], [28] guarantee causality by *statically binding* each client with one of many *DCs*, each of which contains one full replica of the dataset. A simple extension to support partial replication is to treat groups of (geographically close) *DCs* as a single *consistent-hashing ring*, with one replica per object in each ring. For example, Figure 3 depicts eight *DCs* clustered into three rings, with each object having three rather than eight replicas. We consider such a system, which we call *COPS-PR*, as our baseline for comparisons. However, COPS-PR faces a fundamental challenge. Current causal systems require strong consistency within each ring (except [2], which does not address partial replication, as we discuss in Section 7). When a ring spans multiple, geographically-distributed *DCs* as with COPS-PR, strong consistency, availability and partition tolerance cannot be simultaneously satisfied [20]. As such, one unreachable *DC* may render the entire system unavailable because the *DC*'s data is unavailable to the clients bound to the *DC*'s ring.

One may think that the above problem can be fixed by accessing the unavailable data on a different ring. However, the single-ring binding is central to achieving causal consis-

tency in current systems. To see why, consider two objects X and Y that are each present in two rings with initial values, X_{old} and Y_{old}. A client's new values – X_{new} and Y_{new}, in that order – propagate to the two rings independently. If the single-ring restriction were not enforced, another client may read Y_{new} from one ring and the causally-earlier X_{old} from another ring even though causal order within each ring is maintained.

The single-ring restriction degrades availability and latency. First, an object is unavailable if the replica in that ring is not reachable due to network partition or a failure of the *DC* hosting the replica, even though replicas in other rings may be reachable. Second, a client is constrained to accessing a replica in its associated ring, even though a replica in another ring may offer lower latency due to transient network congestion.

Our contributions: In this paper, we present Karma, the first practical system that ensures causal consistency for partitioned data stores with the cost advantages of partial replication. *Karma* employs two novel ideas:

• First, unlike previous causal systems, which statically bind a client to its associated *DC* (or ring), *Karma* allows a client to be served by any replica based on availability or latency. *Karma* leverages the key observation that causality is violated only in the time window from when a causally-later value is visible (Y_{new}) until the causally-earlier value (X_{new}) is propagated to all the rings (i.e., becomes "stable"). Specifically, reads from multiple rings may be inconsistent only in this short window (e.g., 300-400 ms is typical for the geo-distributed 8 *DCs* in Amazon's AWS). Accordingly, *Karma* temporarily restricts reads from a client to go to the same ring as a previous read to an "in-flight" (i.e., as-yet not stable) data object. Because each ring is updated in causal order (like the previous systems), this restriction guarantees that later reads obtain consistent values. *Karma*'s *dynamic ring restrictions* (*DRR*) dynamically tracks in-flight objects to put the threads reading such objects into the restricted mode and to release the threads to the unrestricted, full-choice mode when the objects becomes stable. Because this restriction is transient, *Karma* mostly retains the choice of accessing any ring. Finally, because *Karma* allows ring-switching, it avoids the unavailability problem that may arise when *DCs* are not reachable.

• Second, *Karma* is the first system to integrate causal consistency across persistent *DC*-level storage caches and replicas. Integrating consistency guarantees across the storage and caching tiers is one of the key challenges preventing adoption of stronger consistency models [1]. While all accesses go to the local *DC* in full replication, many accesses go to remote *DCs* in partial replication (and in *Karma*). To achieve low latency with partial replication, it is natural to employ both read caching and temporary, persistent write buffering at each *DC*. Write buffering and caching each pose their own consistency challenges. To avoid consistency problems due to the write-buffer (WB), (1) we use thread-private WBs to prevent the premature reading of values by other threads (which can violate causal-order write propagation), and (2) we require client threads to check their own WBs to see if `gets` can be satisfied from the WB before reading from the cache or storage ring to avoid missing own writes. Similarly, the cache poses a consistency challenge because it may miss for some objects (unlike storage rings which are guaranteed to be complete). For example, a client's cache fill (upon a miss) bringing in the in-flight Y_{new} to a cache that holds X_{old} can violate consistency because (1) the same or (2) a different client may read Y_{new} followed by X_{old}. For the first case, we extend *Karma*'s *DRR* to force the clients, whose read misses return in-flight values, to incur cache misses temporarily for all the reads in the in-flight window. For the second case, *Karma* allows demand fills only with stable objects and not in-flight objects (the cached stable objects are invalidated in causal order as part of write-propagation). These two simple strategies – forced temporary cache misses and disallowed demand fills – differ from conventional caching which does not force misses nor disallow demand-fills and are fundamental to ensuring *Karma*'s practicality.

We implemented *Karma* as a shim-layer between a key-value storage tier consisting of individual (unmodified) Cassandra instances and a YCSB client layer. Experimental evaluation with 64 server nodes emulating 8 geo-distributed data centers in 3 rings shows that *Karma* achieves 43% higher throughput on average and significantly lower read latencies than COPS-PR, while incurring similar costs. Note that *Karma* achieves lower performance than impractical full replication schemes where all accesses are local. However, that is not a specific weakness of *Karma*; rather it is innate to any partial replication scheme. Further, *Karma* offers significantly stronger availability guarantees under failure, and better performance network congestion than COPS-PR. Finally, despite only partially replicating data, *Karma* guarantees full availability under a single *availability zone* [21], [31] failure, and many common network partition modes.

2 Background and Opportunity

We discuss background and *Karma*'s rationale.

2.1 Consistency in cloud storage

Among the consistency models in cloud storage systems are those limited to per-object guarantees. At the weak end of the consistency spectrum are flavors of "eventual consistency" wherein the system offers no guarantees other than eventual propagation of updates to all copies. Eventual consistency may not even guarantee read-your-own-write ordering guarantees; a thread may write a new value and then read an older value. There also exist consistency models which offer stronger per-key ordering guarantees [11], [15], [19], but without any across-key guarantees (which is the focus of this paper).

At the strong end of the spectrum, linearizability offers global ordering of all reads and writes across all keys. However, it is well known that these strong consistency guarantees come at the cost of availability and/or partition tolerance (the CAP theorem [20]).

2.2 Causal Consistency

Causal consistency is stronger than eventual consistency with certain across-key ordering guarantees; yet it is can

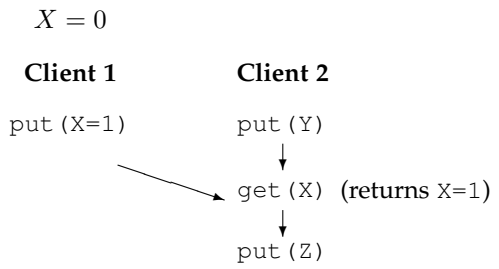


Fig. 1. Inter and Intra thread causal dependencies

achieve both consistency and availability under partition. Causal consistency is a model wherein the global ordering of operations respects each thread's program order as well as the (transitive) ordering implied by any cross-thread value communication [4], [16], [27], [29]. For the special case of concurrent writes to the same object, the writes must resolved deterministically, which can be achieved using version numbers as in previous work [27]. In other words, causality defines a happens-before partial order among `puts` (writes) and `gets` (reads). In this paper, we use the notation $X \rightsquigarrow Y$ to imply that X happens-before Y. As is intuitive, the happens-before partial order is transitive. A causality violation occurs when two operations are perceived to violate the happens-before relationship. Causal systems track causal dependencies to ensure that reads and writes cannot occur in an order that violates causality.

The basic primitive to enforce such ordering in distributed storage systems (assuming a single copy of each object) is "`put-after` " [27]. This operation ensures that causal ordering is enforced in each ring by initiating causally-later writes only after causally-earlier writes are completed even though the items involved may be stored on different servers (or *DCs*) in that ring. For example, in Figure 1, `put Z` is initiated only only after both `put X` and `put Y` complete, though X, Y and Z may be stored on different servers.

The updates occur in causal order in each ring, but proceed asynchronously across the rings. While this ordering provides consistency within each ring, causality may be violated by reading from different rings, as discussed in Section 1. For this critical reason, all current implementations statically bind clients to rings. Recall from Section 1 that such static binding incurs availability and latency problems. While the latency problem is intuitive, one may think that the availability problem can be addressed by chained replication (CR) [37]. CR is appropriate within *DCs* to ensure individual server availability, but does not protect against *DC* failures. However, using CR (which offers linearizability) across *DCs* in the wide area would result in prohibitively slow writes.

In the remainder of this paper, we assume a key-value store that allows `puts` and `gets` on individual tuples. We do not explicitly consider complex key-value store operations such as read-modify-write as they can be interpreted as `puts` for consistency purposes. Transactional atomicity is orthogonal to causal consistency which deals with ordering. Note that general transactions that include both reads and writes are ruled out in a wide-area setting because of the CAP theorem. Some previous papers on causal consistency have also examined limited forms of transactional support (e.g., read-only [16], [27], [28], and write-only [28]) in addition to causal consistency as their motivating examples require both atomicity and ordering. Because ordering is important on its own accord (as illustrated by our examples), we focus on causal consistency. However, we show later in Section 4.5 that *Karma* can support read-only get-transactions by adopting the approach from prior work [28].

2.3 *Karma*'s opportunity

Karma's opportunity arises from the key observation that statically binding clients to rings, as in current systems, is sufficient to ensure causal consistency; but is not necessary.

To illustrate this point, consider the two states in which any object may be. If an object has been written to (using a `put`) and the write is complete (i.e., all replicas have been updated with the latest value) then the object is in a *stable* state. If one replica of an object has been written to (and the asynchronous updates of the other replicas are in progress) the object is in an *in-flight* state.

When a client reads an in-flight value, the client is vulnerable to causality violations because causally-earlier writes may not yet have been applied to all the replicas; so the client may later read a stale value from the not-yet-updated replicas. However, a client that has read only stable values (or has read in-flight values that have since become stable) can not violate causality. This claim follows because all causally-earlier updates must necessarily be complete because of causal-order write-propagation in any given ring. The window of vulnerability is transient, presenting an opportunity for *Karma* to give clients mostly the unrestricted choice of reading from any replica and temporarily restrict later reads to the ring from which an in-flight value was read previously. Because each ring is updated in causal order, the chosen ring is guaranteed to provide causally-consistent values to later reads. *Karma* applies this restriction only during the windows of vulnerability (i.e., until the in-flight value becomes stable), as mentioned in Section 1.

3 *Karma*: Design Overview

Since partial replication results in remote accesses which can hurt latency, *Karma* attempts to minimize remote accesses via the use of per-*DC* caches and persistent write-buffers (WBs). While the latency improvements from caches and WBs are attractive, the challenge of using these multiple tiers while preserving consistency must be addressed carefully. *Karma* ensures that there are no ordering violations as values flow through the WBs, storage rings, and caches, as we describe next.

Karma's goal is to achieve causal consistency by ensuring that the no causally older value may be read *after* a causally newer value has been read. Consider two `put` operations to objects X and Y which previously had the values X_{old} and Y_{old} and which are updated by the `put` operations to have the values X_{new} and Y_{new}. If there is a causal dependency between the two `put` operations with $X_{new} \rightsquigarrow Y_{new}$ (say),

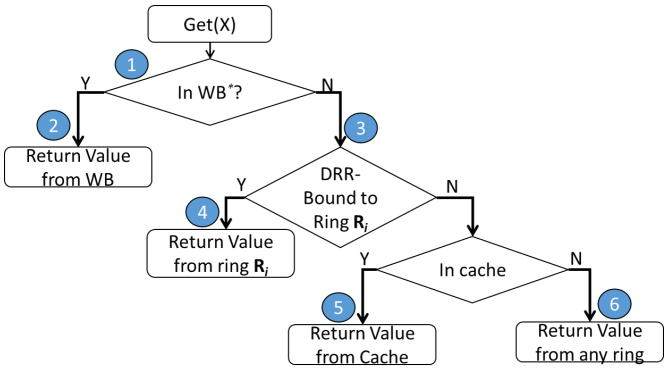


Fig. 2. 'Get' operation in Karma

then, *Karma* (any causally consistent system) must ensure that no client can read Y_{new} and then read X_{old}. *Karma* achieves this overarching invariant by performing `puts` and `gets` as outlined below.

Write Operation: Newly written values enter the WB where values are held in thread order. The values are asynchronously propagated to the storage rings. Like prior causal systems, *Karma* requires causal-order write propagation across rings. This ensures that in any given ring, X_{new} is stored before Y_{new} is stored. As part of write propagation to a ring, *all* the ring's cached copies of the object are invalidated before writing to the ring.

The `get` operation is performed as shown in Figure 2. To understand `get` operation, we consider three cases based on where objects are read from. In each case we show that *Karma* ensures that causality is maintained.

Read Case 1: Objects are read from the WB. Because values in the WB are invisible to other clients, there can be no other causally-newer values outside the WB. Because causally-newer values may be present in the read client's own WB, reads first check the WB before looking in the caches and/or storage rings (see step 1 in Figure 2). (The check in the WB is not as simple as testing presence; we present this detail later in Section 4.1.2.) In our example, if Y_{new} was read from the WB, then either X_{new} will also be read from the WB (if X_{new} has not been propagated from the WB (step 2 in Figure 2), or X_{new} will be read from a storage ring or a cache (if X_{new} has propagated to the ring or cache – steps 4, 5, and 6 in Figure 2).

Read Case 2: Objects are read from the storage ring. In the storage rings (i.e., if the object is not in the WB), there are two cases to consider (step 3 of Figure 2). In the first case, a client thread reads Y_{new} after the value has been propagated to all rings. In this case, causal-order write propagation ensures that X_{new} was previously propagated to all rings. Thus, the client may read X from the cache (step 5) or from any ring (step 6) and is guaranteed to see X_{new} or newer values. As such, *Karma* looks in the cache, and serves the object from the cache (step 5) if it is a hit and from any storage ring (step 6) if it is a miss.

In the second case, a client threads reads Y_{new} from the i^{th} ring R_i (say) before the value is fully propagated to all other rings. In this case, the client thread becomes DRR-bound. *Karma*'s DRR forces reads to access values only from ring R_i. When accessing the ring, because Y_{new} was propagated to ring R_i in causal order, any causally older values (including X_{new}) are guaranteed to be present in ring R_i (step 4).

Read Case 3: Objects are read from the cache. Caches can pose consistency problems if they allow causally-later values of some objects to be brought into the cache while there are earlier values of other objects present in the cache (e.g., Y_{new} and X_{old}). Mixing of old and new values can occur either in the cache (first case) or by accessing some objects in the cache and others in the storage rings (second case).

For the first case, new values may enter the cache through a traditional demand-fill where an object is brought into the cache upon a miss. To prevent mixing of new and old values through demand fills, we disallow the caching of in-flight values that are brought in on demand fills. Thus, the caches hold only stable values which are invalidated upon writes (in causal order), preventing mixing of causally-earlier and later values in the cache. In our example, if Y_{new} becomes stable and is brought into the cache, then X_{old} is guaranteed to have been invalidated. Note that the disallowing is only during the in-flight window and does not prevent caching in the common case.

We address the second case by forcing temporary cache misses during DRRs which ensure access to the DRR-constrained ring, preventing mixing accesses to the cache and to the storage rings (step 4 in Figure 2). In our example, if an in-flight Y_{new} is read by a client, the client is put under DRR forcing cache misses and forcing reads to the DRR-constrained ring which is guaranteed to have X_{new}. These forced misses are only under DRRs which are temporary (during in-flight windows) permitting the benefits of caching the vast majority of time.

In each of the above cases, *Karma* guarantees that it is impossible to read X_{old} after reading Y_{new}. In the next section, we describe *Karma*'s implementation to achieve the operational behavior described above.

4 *Karma*: Implementation

For ease of exposition, we first present *Karma*'s dynamic read restriction without caches in Section 4.1 and then add caches in Section 4.3. These sections assume fault-free operation to focus on *Karma*'s consistency mechanisms. In Section 4.4, we describe *Karma*'s fault-tolerance mechanisms and guarantees.

4.1 Dynamic ring binding in *Karma*

Recall from Section 1 that *Karma* dynamically tracks in-flight objects to put the storage clients reading in-flight objects into the restricted mode and to release the clients to the normal, full-choice mode when the objects becomes stable. Because objects become stable when the corresponding write completes globally (i.e., in all the replicas), detecting global write-completion is a key functionality of *Karma*. In contrast, prior causal systems enforce static client-ring binding which requires detecting only local write-completion (i.e., in the local ring). *Karma*'s other key functionality is dynamic read restriction. Accordingly, we describe in Section 4.1.2

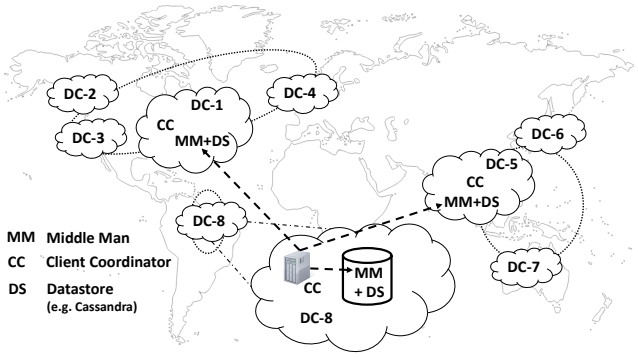


Fig. 3. Karma Architecture Overview

how *Karma* tracks objects' `in-flight` state to detect write-completion; and in Section 4.2 how *Karma* imposes temporary read restrictions.

4.1.1 Basic architecture overview

Figure 3 illustrates *Karma*'s organization. We use any standalone key-value data store (*DS*) at each node. We assume that the geo-clustered sets of *DCs* form one consistent-hashing ring[1] holding one full replica set of the data. In Figure 3, there are three rings, one for each of the US and Western Europe, Asia and Australia, and Brazil; and the Brazilian ring is magnified to show some details discussed below.

Karma requires per-client state (to track causal dependencies) and per-object state (to track `in-flight` versus `stable` status of individual objects). Karma employs a *client coordinator* (*CC*) to redirect client requests to the appropriate back-end servers much like other datastores including non-causal datastores such as Cassandra. We augment *CC* with the additional responsibility of tracking per-client causal meta-state. There can be multiple *CC*s per *DC*. The *CC* is responsible for two major tasks. First, it is responsible for causality-preserving write-propagation to all rings from the write-buffers and for satisfying the safety property of detecting write-completion. Second, the *CC* enforces temporary restrictions to ensure that causality is not violated in the window of vulnerability (Section 4.2).

To track the per-object `stable` versus `in-flight` state, one may either provision per-object state (1 bit/object) or equivalently, use a set of inflight objects. We introduce a module in the storage layer called the *middle man* (*MM*) which holds per-object metastate; there is an *MM* for the replica in each ring. Figure 3 shows a *CC* in Brazil interacting with an *MM* for an object's replica in each of the three rings. The *MM* and storage server can be co-located on the same node so that the *MM* holds the metastate for the data shard on the server. To prioritize modularity and separation of concerns, we implement the *MM* as a separate module isolating causality-related metastate from the underlying datastore, though one could alternatively implement the *MM* as an integral part of a causally-consistent datastore.

1. *Karma* extends to directory-based object placement by having the set of i^{th} replicas in each directory entry form the i^{th} ring.

Together the *MM* and *CC* guarantee the safety property for causality enforcement: a given version of an object must be considered stable only if the version (or a later version) of that object is present at *all* the replicas (rings). Safety is not violated if a stable object is considered temporarily to be in-flight.

4.1.2 Global tracking of writes and detection of write completion

Every `put` starts with a write to a client-private persistent WB, after which the client is free to proceed. *Karma*'s causal order propagation of writes from the WBs to the rings uses `puts` to the storage rings, orchestrated by the *CC* and *MM* as in Figure 4. The *CC* appends any received `puts` to the WB and to the tail of per-ring propagation queues (ENQ in Figure 4). A per-ring propagator thread in the *CC* processes entries in queue order and uses `put` or `put-after` to propagate values to their corresponding ring. In Figure 4, the propagator threads, *PT1*, *PT2*, and *PT3*, propagate the values to *MM1*, *MM2*, and *MM3*, respectively. As in the previous causal systems, each propagator thread propagates values to its ring at its own pace, not synchronizing with the other propagator threads.

Similar to Orbe [16], *Karma* achieves causal-order write-propagation from the front-end *CC*. This eliminates the within-thread `put-afters` by using client-side program-order write-propagation. For example, for the scenario in Figure 1, the system generates a `put-after` at *client 2* to ensure that `put Z` occurs after `put X` in *client 1* to enforce the *inter-thread* dependence induced by the `get X`. However, because `put Z` and `put Y` are in the *same* thread (*client 2*), no `put-after` is needed if the `puts` are completed in thread program order. This is distinct from COPS [27] which does write-propagation not from the front-end client but from the storage server where all thread-program-order information is already lost. Consequently, COPS uses `put-afters` even for within-thread ordering. As a further optimization over Orbe [16], *Karma* only includes items that are in-flight as part of its `put-after` dependencies, since stable items are known to be written.

Upon receiving a new `put`, the *MM* in each ring transitions the object to the `in-flight` state (by including it in the `in-flight` set). After the local `put` is complete, the *MM* sends an acknowledgment to the *CC* that initiated the request. The *CC* marks the write as propagated to that ring. Tracking the propagation to each ring is also useful when determining whether a value can be forwarded from the WB or not. Specifically, when a `get` request looks up the WB while under *DRR* (bound to ring R_i, say), if the object is already propagated to ring R_i, the WB lookup fails.

The completion of the `put` on the local ring of the object does not guarantee the object is stable. Rather, the last propagator thread to process the per-ring `put` essentially triggers the *CC* to detect global write-completion. By waiting for all the propagator threads, the *CC* detects global write completion despite asynchrony of write propagation across rings. The *CC* sends a notification to each *MM* which marks its object copy as stable (in Figure 4, see "WRITE COMPLETE" on the left and "STABLE" on the right). The

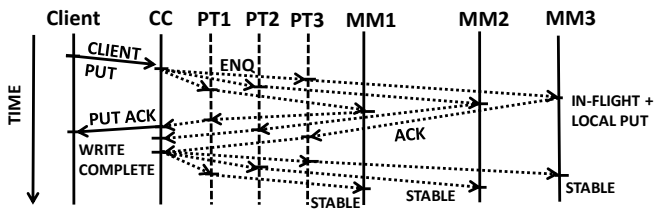


Fig. 4. Write-propagation in *Karma*

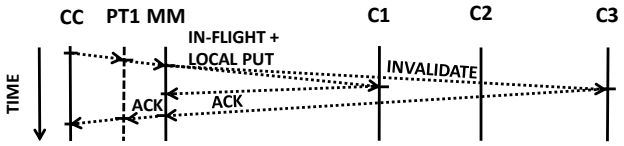


Fig. 5. Write-propagation in *Karma* with caches

CC also evicts the object from the WB. Thus, the *CC* and *MM* achieve the safety property of detecting write completion.

4.2 Dynamic causality enforcement

The *CC*'s second major task is enforcing temporary read restrictions to ensure causality. Recall from Section 2.3 that clients that read (via `gets`) stable values are not vulnerable to causality violations. As such, these clients operate in the unrestricted mode which is the common case. For such clients, *Karma* is free to route the `gets` to *any* ring based on availability and network proximity.

In the uncommon case, when a client reads an in-flight value from a ring (as indicated by the *MM*), the *CC* dynamically restricts the client to that ring by using a per-client *Dynamic Ring Restrictions* (*DRR*) structure. Recall from Section 2.3 that because each ring is updated in causal order, this restriction ensures that the client's later reads obtain consistent values. *DRRs* are tied to an object and a version number to avoid premature transitions to the unrestricted mode when there are multiple restrictions. For example, a client may read multiple in-flight values (or read multiple in-flight versions of the same object). Such a client must wait for all the restrictions to be lifted before returning to the unrestricted mode. Nevertheless, the restrictions are short-lived due to relatively fast write-propagation (e.g., 300-400 ms for 8 geo-distributed *DCs* in Amazon's AWS under no-load conditions).

Lifting the *DRRs* poses the interesting challenge that the *CC* associated with a read of an in-flight object is not notified of the object's write-completion. Only the *CC* that originates the write and the MMs responsible for the object in each ring are made aware of write-completion (Section 4.1.2). Without additional safeguards, a *DRRs* would become permanent upon the first access to an in-flight object (and degrade to static binding). A naive approach of maintaining per-object state, that tracks the reader *CC*s for notifying write-completion, would be cumbersome and would incur significant tracking overhead. Fortunately, there is an elegant way to capture write-completion without additional effort. We exploit the fundamental transitivity property that *whenever a restricted client performs a* `put`*, the completion of that* `put` *guarantees the completion of any earlier* `put` *(from any client) that may have been read by the client*. Because (1) writes within each ring are done in causal order (via `put-after`) and (2) the *CC* detects global write-completion across all the rings, a client's `put` can complete only after all causally-earlier `puts` from any client are complete (e.g., in Figure 1, `put Z` can complete only after `put X`). Consequently, each `put` completion removes all restrictions due to earlier `gets` in program order without reader *CC*s receiving write-completion notifications.

Of course, one must also consider read-only clients where such natural garbage collection of expired restrictions does not occur. To avoid permanent read restrictions for such clients, we propose *write insertion* which inserts dummy writes periodically for the purpose of such garbage collection via `notify-after`, a new primitive. `notify-after` returns write-completion notification after verifying the same dependencies as a `put-after` would, but without actually writing to the *DS* (and hence avoiding the full overhead of a write). One can tune the frequency of the `notify-after`s to balance their overhead and removal of expired *DRRs*.

4.3 Write-completion with caches

Write-propagation in the presence of caches is organized as a two-step mechanism. In the first step writes are propagated from the *CC* to the *MM* as described in Section 4.1.2. The *MM*'s are responsible for write-propagation to the caches, which is the second step. Each *MM* holds a *cached-at* set per object which tracks the *DC* caches that obtained the object from the *MM*. Because no single *MM* may naturally know the location of all the cached copies, each *MM* tracks only a subset of *DCs* where each object may be cached. The *MM* conservatively assumes that any *DC* that accesses stable objects may cache the object; and thus adds the *DC* to its *cached-at* set for that object. *DC* caches may evict objects from the cache without notifying the *MM* (i.e., silent replacement). Upon the next write, the MM will send an unnecessary invalidation and remove the cache from the list.

The second step of write-propagation from the *MM* to the caches may be achieved via updates or invalidations. While there are well-known tradeoffs in using either of updates or invalidations, the latter are simpler and hence our choice. After sending invalidations to the caches, the *MM* must wait for acknowledgments from the caches before sending its acknowledgment to the propagator threads, as discussed in Section 4.2. This waiting ensures the safety property of detecting write completion (Section 4.1.2). Figure 5 shows the acknowledgments from the caches (*C1*, *C2*, and *C3*). The invalidations ensure causal-order write propagation to caches. Instead of invalidations, cached data may also be self-invalidated via leases that expire after a time-out. Writes would complete faster in this approach (no invalidations needed) implying fewer *DRRs*, at the potential cost of unnecessary discarding and refetching of valid data.

4.4 *Karma*: Fault tolerance

We have designed *Karma* to be available under server failures, failure of a single availability zone (AZs) [21], [31], and a broad class of network partitions. Upon network partitions or an AZ failure, *Karma* remains available but may operate in a degraded mode wherein causal consistency is guaranteed, but dynamic ring switching remains prohibited till all zones are up (or till all partitions are healed). To put that in perspective, (a) *Karma*'s common-case, fault-free performance is better than that of COPS-PR because of dynamic binding, and (b) though *Karma* suffers from static binding under an AZ failure, COPS-PR is not even available. In fact, *Karma* (under failure) incurs the static binding penalty that COPS-PR always incurs (even when fault-free). While *Karma* is not guaranteed to be available under multiple, simultaneous AZ failures, such failure modes are relatively rare.

We assume a reliable transport (e.g., TCP, or application-level ack/retry mechanisms). Table 1 summarizes *Karma*'s resilience under common failure modes, compares *Karma* with other schemes and lists *Karma*'s mechanisms. We now discuss individual failures:

• *Individual* MM*/Backend-server/rack failure:* Since the *MM* is co-located with the storage server, *Karma* leverages Chain Replication (CR) [37] within the same *DC* to protect both data and metadata (in-flight status and *cached-at* sets) against individual server or rack failures (assuming CR spans multiple racks).

• *Cache node failure:* Because *Karma* caches only stable state, the cache state can be re-fetched if lost. Loss of cached data can never cause correctness problems (e.g., consistency violations or unavailability); it may, at worst, lead to performance penalties of remote data access. Also, a cache node failure means invalidations, and hence writes, to the cached data cannot complete potentially causing some *DRRs*, and hence performance penalties, but no availability problems.

• *CC failures (individual and* DC*):* Failures involving the *CC* requires more careful treatment since the *CC* is responsible for write propagation. Liveness of write propagation is key to achieving write completion which is central to *Karma*. It is relatively easy to protect against individual *CC* failure, by applying within-*DC* chain replication for *CC*'s as well. However, if the entire *DC* containing a *CC* fails (or is partitioned from the rest of the world), then those writes will indeed stop propagating. A client that is unaffected by the failure, and is *DRR*-bound to the same ring as the failed *DC* which contains the write's *CC* (because of reading the write before the failure), may be indefinitely bound to the ring with the failed *DC*. Data in the failed *DC* is no longer available to the client.

We observe that this corner case occurs because of a correlated failure where both *CC* nodes and backend servers fail. To prevent such correlated failures, we simply require that the front-ends (including *CC*s) and back-ends (storage nodes) reside in different availability zones. Note that *DC*s are typically architected using multiple availability zones, which are isolated from each other and connected through low-latency links [21], [31]. We refer to availability zones with front-ends and back-ends as *AZ*-F and *AZ*-B respectively.

As such, though write-propagation has stopped due to the *CC*'s (*AZ*-F) failure and the client is *DRR*-bound to the *AZ*-F's ring (due to reading an incomplete write), the complete data is available in the ring (the *AZ*-Bs are operational). On the other hand, if an *AZ*-B fails, then write-propagation of the *DRR*-causing object (i.e., the in-flight object which the client read and got *DRR*-bound) and all its dependencies will complete because (1) the writes of the object (and its dependencies) have completed in the ring to which the client is *DRR*-bound (by definition, since that caused the *DRRs* originally), and (2) the writes are guaranteed to propagate to the other rings (*AZ*-B in other rings and all *AZ*-Fs are alive). Therefore, the *DRR* will lift allowing access to the other rings with the complete data.

Resilience Analysis: We discuss the resilience of *Karma* to *AZ* failures, and network partitions. We define a ring to be completely-available if all *AZs* in that ring are reachable from all clients. We make two observations:
(1) A client C has no availability problems if (i) there is at least one completely-available ring; and (ii) C has no *DRRs*, or is *DRR*-restricted to the completely-available ring.
(2) A client C has no availability problems when it is *DRR*-restricted to a ring with a failed or unreachable *AZ*, provided all other rings are completely-available.

(1) is trivially true. To see (2) consider that upon a *AZ*-F failure, the ring to which the client is *DRR*-bound has the complete data. And upon a *AZ*-B failure, write propagation of the *DRR* causing object and its dependencies completes, allowing the *DRR* restriction to be lifted (as argued for the corner case in Section 4.4). C may then access other rings with no availability problems. These observations enable the following claims:
*Claim 1 [*AZ *failures]*: *Karma* ensures availability under the failure of an arbitrary number of *AZs* of the same type in one ring (i.e., either *AZ*-F's or *AZ*-B's), provided all *AZs* in all other rings are completely available.

This claim holds because a client which is *DRR*-restricted to the ring with failures has no availability problems as per (2). In all other cases, (1)holds.
Claim 2 [Partitions]: *Karma* ensures availability
(1) [*intra-ring*] under any intra-ring partition isolated to a single ring provided *AZs* in all other rings can reach one another, and there are no inter-ring partitions (i.e., *AZs* in different rings can reach each other)
(2) [*inter-ring*] under an inter-ring partition where a pair of *AZs* in two different rings are unreachable from each other, but *AZs* in any other pair of rings can reach each other, and there are no intra-ring partitions.

The proof for case 1 follows an identical argument to Claim 1 above. For case 2, assume the partition occurs between rings *R1* and *R2*. If both *AZs* are of the same type (both *AZ*-Fs, or both *AZ*-Bs), the partition trivially poses no problem because data in all the rings is completely available to all the clients. Consider that a *AZ*-F in R1 is partitioned from a *AZ*-B in R2. For a client in $R1$, because all the rings except *R2* are completely-available, there are no availability problems unless the client is *DRR*-restricted to *R2*. However, by Observation 2 this *DRRs* does not lead to availability problems.

TABLE 1
Impact of failures (†includes COPS/Orbe/Eiger, 'Better*' implies *Karma* is better than COPS-PR)

Failure	Available?	Contrast to		Protection Mechanism
		Full Replication†	COPS-PR	
Backend Server	Yes	Same	Same	Chain replication
Cache Server	Yes	Not applicable	Not applicable	Stable state
Rack	Yes	Same	Same	Chain replication
CC server	Yes	Same	Same	Chain replication of *CC*
Single *AZ*	Yes	Same	Better*	Dynamic binding
AZ-B Cutoff	Yes	Same	Better*	Dynamic binding
AZ-F Cutoff	No	Worse*	Same	*Partial replication limit
Partition	Yes	Same	Better*	Dynamic binding

In summary (Table 1), *Karma*'s flexibility of ring switching leads to (a) better availability than COPS-PR which employs static-ring binding, and (b) similar availability as full-replication systems which incur much higher cost than *Karma*. Note that there are scenarios where full-replication systems achieve better availability. For example, if one *AZ*-F gets completely network-partitioned from the rest of the world the clients within that *AZ*-F would see unavailability under all partial-replication schemes including *Karma*, independent of the consistency model or system, but would have no availability problems with full-replication systems. Even so, *Karma* ensures all other clients do not see unavailability.

4.5 Get-Transaction Support in Karma

Recall from Section 2 that the general case of read/write transactions are not appropriate for our domain because our goal is to offer consistency and availability even under partition. The serializable semantics of read/write transactions is ruled out because of the CAP theorem. However, read-only transactions (get-transactions), which effectively offer the ability to group a collection of `gets` such that they read an instantaneous snapshot without any intervening writes is be achievable (i.e., not ruled out under the CAP theorem) [16], [27], [28].

While our focus in this paper is on ordering which is orthogonal to transactional atomicity, we show that *Karma* can leverage prior designs to support read-only transactions. Specifically, Eiger's [28] read transactions are based on (1) eagerly attempting to read the latest values, (2) detecting if such reads form a snapshot without intervening writes, and (3) if the reads are determined not to be a snapshot, reconstructing a snapshot by reading appropriate older versions (which must be saved). There are additional optimizations to avoid indefinite retention of older values and to abort/retry transactions if they cannot complete within specified timeouts.

Karma can use the same get-transaction mechanism used in Eiger with only two minor changes to address dynamic ring binding and caching.

Interaction with Dynamic ring binding Eiger's get-transactions mechanism works within a single statically bound ring/DC. Similarly, in *Karma*, all transactional reads go a single ring. However, Karma can choose a different ring for each transaction if there are no *DRRs*. Even within a transaction, if the transaction aborts (because of a timeout/failure), Karma can retry the transaction in a different ring. This last feature is important for Karma because if some data is not available in a ring, the transaction can be reattempted at another ring.

Interaction with Caching Karma can bypass all interaction of transactions with caching by requiring all transactional operations to bypass the cache.

Finally, while it is also possible to incorporate Eiger's write-only transactions in *Karma*, we believe it is not worth the complexity because it involves implementing two phase commit in the wide area.

5 Experimental Methodology

We used a 64-node cluster in the Probe test-bed [18].

Modeling geo-replicated settings: To model geo-replicated settings, we group *DCs* into multiple geographic regions, as shown in Figure 3. We considered eight regions, modeled after Amazon AWS, with three in the US, (2 in the West and 1 in the East Coast), one in Europe, one in South America, and three in Asia/Australia. We measured delay across EC2 instances in different regions and emulated these delays in our cluster using Dummynet [10]. For instance, the round-trip delay from US-West1 to US-East, US-West2, Europe, Singapore, Tokyo, Sydney and Brazil were 86, 23, 175, 221, 143, 198 and 205 milliseconds respectively.

Clustering *DCs* into rings: Our evaluations of *Karma* used three rings, comprising (i) all the *DCs* in the US and Europe; (ii) all *DCs* in Asia and Australia; and (iii) the *DC* in South America. This partitioning generally ensures that within-ring delays are lower than across-ring delays. Because of our target of three rings, Europe's relative network proximity to the US puts their *DCs* in the same ring. *Karma*'s design is independent of partitioning heuristics. More generally, factors besides network proximity may be considered in ring partitioning.

Schemes: We compare *Karma* using the rings described above with the following state-of-the-art schemes. We implemented each of *Karma* and these schemes as a shim-layer between a key-value storage tier consisting of unmodified, individual Cassandra instances and a cloud client layer. The *MM* maintains the necessary metastate for *Karma*, which is minimal. There is a table of inflight objects which is small because there are only a few in-flight objects at any given time. The *cached-at* metastate holds 8 bits per object (a full bitmap of eight *DCs* where the object is cached).

• *COPS-Ideal:* This full-replication scheme replicates all data items in each of the eight *DCs*, and includes key optimizations in COPS and Orbe (Section 2.2). To validate that our implementation of COPS-Ideal is similar to COPS, we measured the achieved throughput of our COPS-Ideal implementation with the same configuration (2 *DCs*, 1 server/*DC*, and zero wide-area delays) as in the original COPS paper. Because our hardware is different from that in the COPS paper, absolute throughput comparisons are not meaningful. As such, we compared throughput as a fraction of peak sustainable throughput of pings between servers. Such ping throughput represents an upper-bound on achievable throughput. Our COPS-Ideal implementation achieves comparable throughput (within 8% and better latency (99^{th} percentile) than COPS.

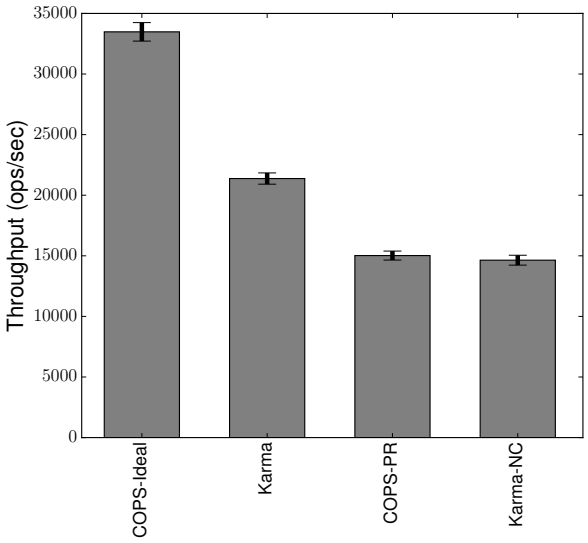
• *COPS-PR:* Recall from Section 1 that this more practical COPS-variant is a straightforward extension of previous causal systems to support partial replication. This scheme uses (1) the same three rings as *Karma* (therefore resulting in identical cost), but with the restriction that reads arriving at a *DC* may access only the replica in the *DC*'s ring; and (2) write buffering for fast local writes but no caching (including which would require the causality-preserving caching techniques of *Karma*).

One may imagine an alternative system with equal cost as *Karma* wherein the three replicas are located in three of the eight *DCs*. The remaining 5 *DCs* will all see degraded latencies and availability on partition; as such, we limit the comparison to COPS-PR which also has the same cost as *Karma*.

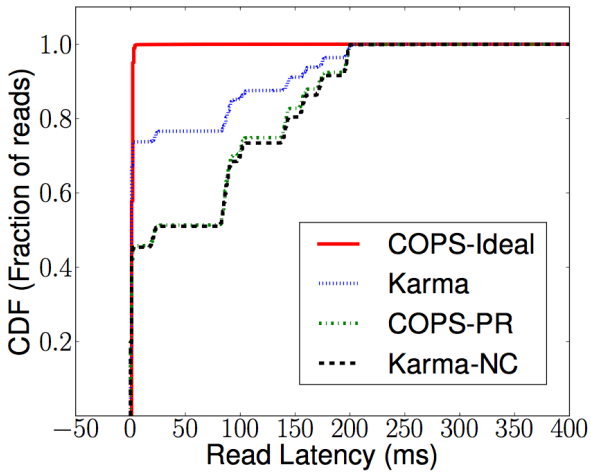
• *Karma-NC:* This *Karma*-variant excludes caches but includes write buffering for fast writes.

Experiment configuration: Each *DC* comprised of eight storage nodes (each of which ran a single-node Cassandra datastore and our *MM* (per-object state) code). The *CC*'s (which hold per-thread state) are also co-located on the same nodes; however, any *CC* may access any back-end. For *Karma*, we considered two cache nodes in each *DC*. To avoid giving *Karma* a resource advantage, we reduce the number of storage nodes in *Karma* to six per *DC*. This paper focuses on achieving consistency and leaves design-space exploration, such as optimizing the number of storage and cache servers, to future work. Further, our performance results include all the overhead of write-completion notifications and invalidations in *Karma* (Figure 4 and Figure 5).

Workloads: We used the well-known *Yahoo! Cloud Serving Benchmark (YCSB)* [12]. We focus on read-heavy workloads with a read-write ratios of 95-5, which is used extensively in prior work [3], [4], [11], [16], [27]. For each configuration, the number of client threads are empirically increased till the system saturates. The number of client threads at saturation for COPS-Ideal, *Karma*, Karma-NC, and COPS-PR, are 600, 1050, 1200 and 1200, respectively. We also include sensitivity analysis for more write-heavy workloads. Each record has 200 bytes of data by default, but we report on sensitivity of our results to object size (Section 6.3). We run seven experiments for each experiment and show the standard deviation to quantify run-to-run variation. For each experiment, we loaded the system with 500 million records, and ran for 10 million operations.



(a) Throughput



(b) Latency

Fig. 6. Throughput and latency comparison.

6 Results

6.1 Performance Results

Figure 6(a) and (b) illustrate the sustainable throughput and `get` latency (`put` latency is omitted as `puts` are always local) achieved with each of the four schemes.

Several observations can be made from Figure 6. First, *Karma* achieves 43% higher throughput on average, and significantly lower latency than the equivalent cost COPS-PR. COPS-PR has lower per-thread throughput than *Karma* because of the absence of caches. However, COPS-PR uses a higher number of client threads (1200 vs. 1050) to saturate the system. Second, the throughput and latency of Karma-NC and COPS-PR are similar as expected. The primary advantage of Karma-NC over COPS-PR is Karma-NC's ability to adapt to failures and network congestion by accessing

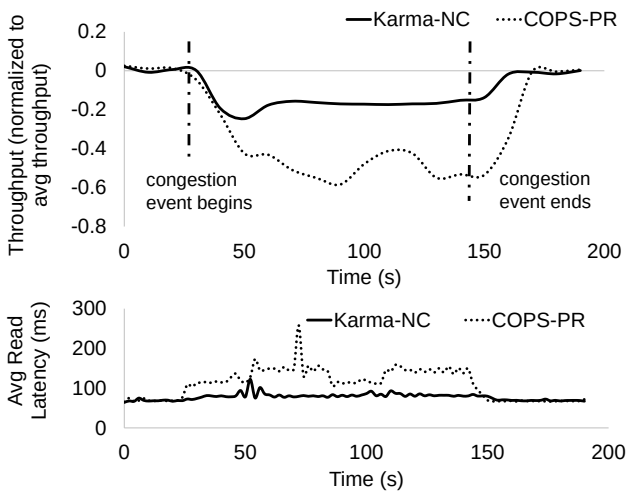


Fig. 7. Throughput and latency under congestion

other rings (Section 6.2). Third, the performance gap (both in throughput and in latency) between the ideal COPS (which is impractical because of full-replication) and *Karma* is because of partial replication. Specifically, (i) COPS-Ideal achieves 100% local `gets` by incurring the high cost of full replication. In contrast, *Karma*'s local reads come from caching, which cannot achieve 0% miss rates. In addition to local reads due to caching, *Karma*'s also benefits from local reads in two other cases: `gets` that are served from the local write-buffers, and `gets` of objects that are mapped to the local *DC* in the storage ring. The aggregate effect is that *Karma* achieves nearly 77% of local accesses. *Karma*'s key advantage over COPS-Ideal is its reduced write propagation costs. However, this advantage is diminished in our read-heavy workload where only 5% of operations are `puts`. Indeed, our sensitivity experiments (Section 6.3) revealed that as workloads become more write-heavy, *Karma* can match and even out-perform the throughput of COPS-Ideal.

6.2 Importance of dynamic ring binding

A key performance advantage of *Karma* is its ability to allow clients to dynamically select replicas from any ring, rather than statically binding a client to one ring. To evaluate the dynamic ring switching ability of *Karma* separately from the caching component, we focus on Karma-NC and COPS-PR.

Specifically, we examine system behavior when subjected to network congestion, and evaluate the effectiveness of Karma-NC in ensuring good performance even in such scenarios. We emulate congestion by sharply increasing the latency (by 300ms, 600ms, and 1200 ms) of all traffic in and out of one randomly chosen DC (Europe). We maintain this congestion for a period long enough for the systems to settle (120s in our experiments), and then revert to the uncongested state. Background processes continually monitor the delays between DCs, and feed the information to the systems.

Figure 7 shows the time-varying behavior (normalized throughput in the top graph and read latency in the bottom graph) of the two systems (the two curves) with the 1200ms added delay in the Europe *DC*. Karma-NC and

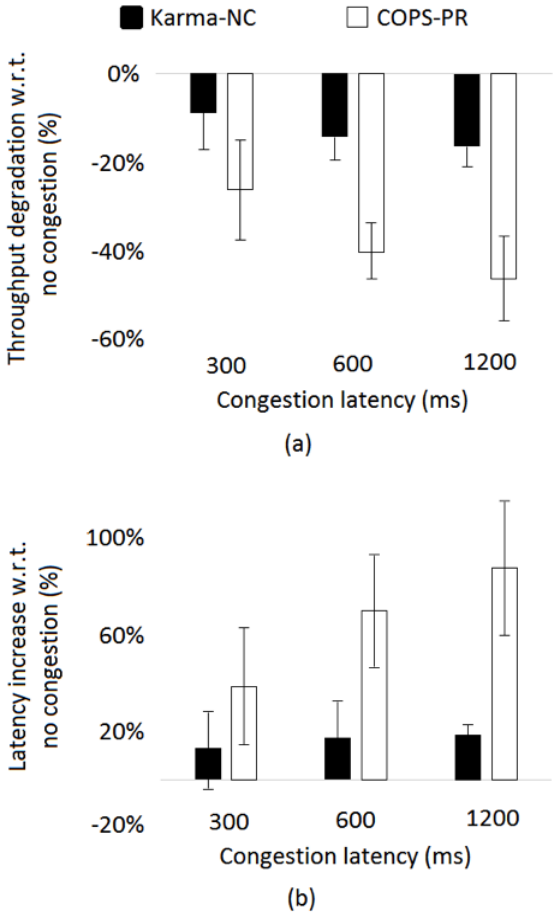


Fig. 8. Throughput and latency under congestion

COPS-PR are similar in performance in an uncongested environment. However, Karma-NC performs significantly better than COPS-PR during the congestion event. This is because the static ring-binding in COPS-PR forces clients in the ring which includes the congested European *DC* to incur the full penalty for all accesses to the Europe *DC*. In contrast, Karma-NC provides clients with the flexibility to access data from other rings rather than incur the high latency of going to the European *DC*. Finally, Karma-NC does see some performance degradation during the congestion event compared to its performance under normal conditions. This is because redirecting accesses to remote rings does involve higher latencies than local latencies under normal conditions. Further, the congestion event extends write completion times of objects that are held in the Europe *DC* which may impact performance.

The two graphs in Figure 8 show the throughput and latency degradation (averaged over the time of the congestion event, relative to uncongested operation in percentage) on their respective Y-axes, for congestion delays of 300ms, 600ms, and 1200ms (X-axis). Karma-NC demonstrates consistently better performance under congestion and degrades at a slower rate than COPS-PR. This is because of Karma-NC's ability to switch to other rings as discussed above.

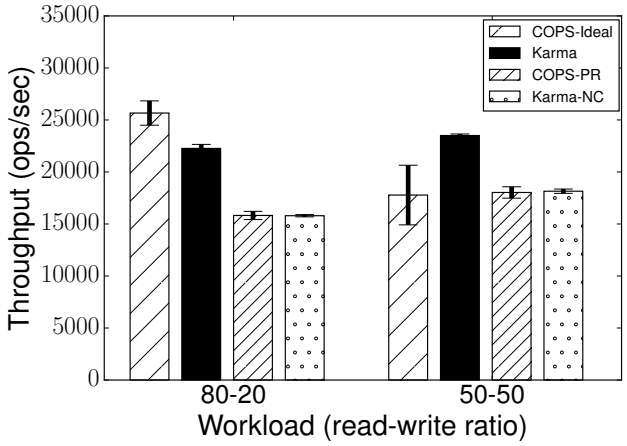


Fig. 9. Sensitivity to read-write ratio

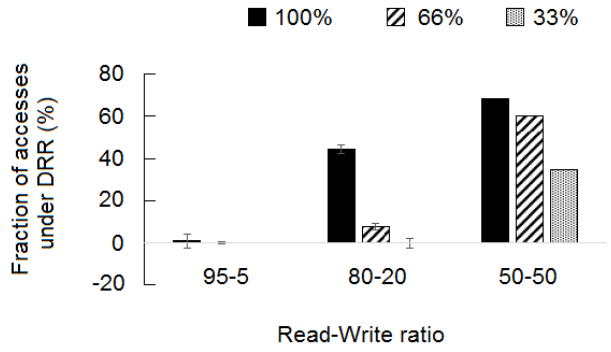


Fig. 10. Impact of workload on access fraction under DRR

6.3 Workload sensitivity and DRR behavior

While our results so far assume a read-write ratio of 95-5, Figure 9 presents throughput for workloads with higher write ratios. *Karma* continues to out-perform the cost-equivalent COPS-PR scheme across read-write ratios. Interestingly, with increasing `put` fraction, *Karma*'s performance improves relative to COPS-Ideal. The performance is comparable at 80-20 (within 13%), while at 50-50, *Karma* achieves 32% higher throughput than COPS-Ideal. This is because COPS-Ideal incurs significantly higher replication cost for each write than *Karma* (8X vs. 3X). Thus, *Karma* not only achieves its primary objective of significantly reducing costs as compared to COPS-Ideal, but also achieves comparable or better throughputs at higher `put` ratios.

Figure 10 shows the fraction of accesses made while under DRR (Y-axis) for different read-write ratios (X-axis). For each ratio, we not only show performance under saturation throughput, but also at 33% and 66% of saturation load since real systems typically operate at a smaller fraction of their peak load. With 95-5 traffic, less than 2% of accesses are under DRR for all load levels. Even under the most extreme datapoint (100% load with 50% `puts`), more than 30% of accesses are not DRR-bound (i.e., they retain the ability to switch rings). In contrast, prior static-binding approaches are always bound to the local ring.

One key concern is that *Karma* has to bypass caches

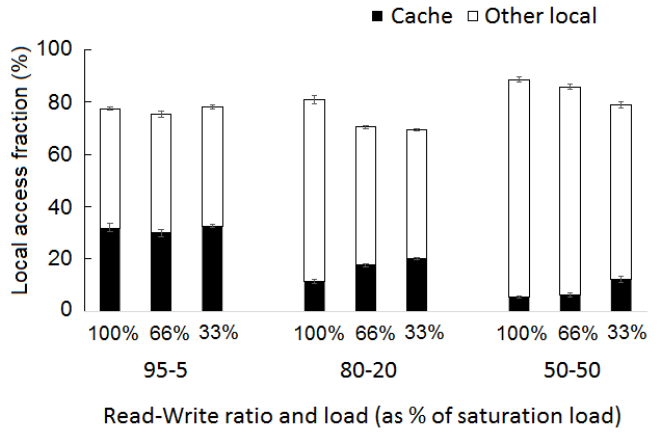


Fig. 11. Local access fraction under varying read-write ratio and load

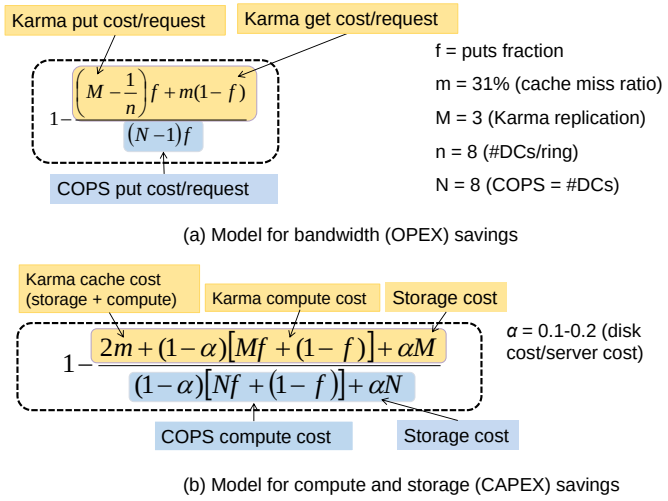


Fig. 12. Models for cost savings with *Karma*

when under DRR. However, we show that the penalty of bypassing the cache is more than compensated by the increasing number of accesses that are served from the write buffer. Figure 11 shows the aggregate fraction of `gets` that are served locally on the Y-axis (i.e., from the cache or from the write-buffer or local DC) for varying load levels and read-write ratios (X-axis). As expected from the *DRR* trends, higher load levels and higher `put` fractions are associated with increasing *DRR* which correspondingly results in lower cache hit ratios. However, there is a compensatory effect due to reads being served from the write-buffer which results in an overall improvement in the fraction of local accesses. This again is not surprising given the popularity skew in typical cloud storage traffic which is modeled as a Zipfian distribution in YCSB.

6.4 Cost Analysis

To quantify *Karma*'s cost savings over COPS-Ideal, we use simple cost models for inter-*DC* bandwidth costs (OPEX) (Figure 12(a)), and compute and storage costs (CAPEX) (Figure 12(b)), and *Twitter* traces [26] to ground our analysis.

We breakdown the storage and compute costs of a single unit cost server as α and $(1-\alpha)$, respectively. We then scale the compute and storage costs independently to match the compute and storage demand. We model compute cost as proportional to the number of local operations (e.g., each `put` generates as many local operations as number of replicas). Further, *Karma* has two local operations for each cache miss (initial lookup + demand fill). Storage cost scales in direct proportion to the degree of replication. For *Karma*'s caches, we assume that storage costs scale the same as compute costs. Finally, we model the inter-*DC* communication costs to include `put` costs (replication cost for every `put`) and `get` costs (misses). Because compute and storage costs are capital expenditures and bandwidth costs are operational expenditures, we treat these costs separately.

Analysis using *Twitter* data: To drive our model with realistic data, we used a single day of publicly-available *Twitter* traces [26] which included a user-friendship graph, a list of user locations, and public tweets sent by users (along with timestamp). We assume every user reads each tweet from a friend exactly once, which yields a 3.5% miss rate for *Karma*. Note that miss rates would be lower if a user accesses the same tweet multiple times.

Figure 13(a) plots the compute and storage cost (CAPEX) savings of *Karma* over COPS-Ideal on the Y-axis for various `put` fractions (X-axis) using the miss ratio obtained from the *Twitter* analysis, and for multiple values of α. Not surprisingly, the cost savings exceed 50% for write-heavy workloads. Even for read-heavy workloads, the cost savings are between 32% and 43% (depending on the storage cost fraction). Even under the 31% miss ratio seen in our 95-5 workload, the cost savings vary from 8% to 26% (not shown).

Figure 13(b) presents the bandwidth cost (OPEX) savings with the *Twitter* trace. The bar on the left (workload-agnostic) shows the cost savings assuming the partitioning of *DCs* into rings described in Section 5. However, this partitioning is not cognizant of the distribution of user locations. In the *Twitter* trace, 70% of tweets were from users in the US East Coast, and 29% of tweets from users in the US West Coast. We considered cost savings with an alternate workload-aware partitioning (right bar) which comprised a single ring for the US East Coast *DC*, a ring for *DCs* in the US West Coast, and another ring for all the other *DCs*. While workload-agnostic partitioning already results in cost savings of 14% in inter-*DC* traffic, a workload-aware partitioning could lead to cost savings of over 30%. More generally, these results point to the benefits of exploiting user location information, when available, in addition to network delays, as part of the partitioning strategy. Further, we have used a simple model of just counting total bytes of inter-*DC* traffic. In practice, shipping data over trans-continental links costs more, and we expect *Karma*'s savings likely to be even more if this detail is factored in.

6.5 Sensitivity to Object Size

We varied object size from 200 B (default) to 1 KB, and 4 KB. We do not consider larger objects like images and videos because causal consistency is relevant only for mutable objects.

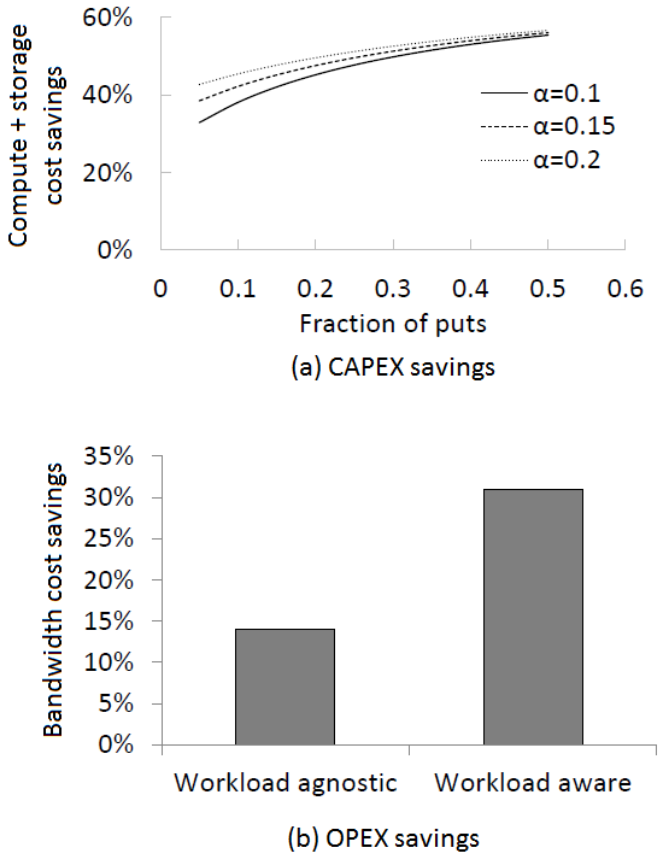


Fig. 13. *Karma* cost savings (over COPS-Ideal)

Karma's performance degrades gracefully to larger objects; the throughput with 5X larger objects (1KB) is within 13% of the default *Karma* throughput (with 200B objects) and within 32% with 20X larger objects (4KB).

7 Related Work

Classical works on causal consistency(*e.g.,* [6], [29]), some of which consider partial replication [6], are limited to single node *DSs*. In contrast, we focus on large *DSs* that span hundreds of nodes. Other recent and concurrent ongoing efforts are investigating partial replication with causally consistent datastores. OPCAM [32], [33], [34] is a proposed causal system with partial replication that is a strict subset of our COPS-PR (it uses static ring-binding). Recall that static ring binding leads to availability or consistency problems (i.e., it is subject to CAP constraints). Other preliminary work on partial replication is either underspecified [14] (e.g., early vision papers, without an implementation or evaluation) or has the static binding problem [7]. *Karma* is the first scalable, causally-consistent data store to support partial replication with or without *DC*-level storage caches while offering consistency and availability under partition.

There are distributed stores that use primary-secondary replication in which all writes are written to the primary replica from where they are propagated to secondary copies (e.g., [9], [36]). Specifically, Pileus shows that causal consistency can be achieved in such a system. However, the design choice of directing writes to the primary replica

results in either unavailability under partition or violation of consistency. *Karma* achieves both availability *and* consistency under partition by not requiring writes to be funneled to primary replicas. In addition, directing all writes to a primary site also hurts write latency. In contrast, *Karma* allows all writes to be local.

ChainReaction [2], which employs a modified variant of Chain Replication within each *DC*, relaxes the requirement of linearizability within the *DC*. However, because of static binding, ChainReaction is still dependent on full replication; with partial replication ChainReaction remains susceptible to unavailability when a *DC* that holds the head of a chain is unavailable.

Bolt-on causal consistency(BOCC) [3], [4] enforces programmer-annotated *explicit* causality and not for all *potential* causality like *Karma*. SPANStore [38] seeks to reduce costs through partial replication, but does not consider causal consistency.

There is interest in supporting transactions in geo-replicated storage systems [5], [13], [22], [27], [28], [35], [40]. Many of these systems (*e.g.*,, [5], [13]) uses Paxos [24] in the wide-area, which is incompatible with low latency, and sacrifices availability under partitions. Causally consistent systems (including *Karma*) which are typically designed for availability under partition usually offer limited forms of transactional support as described in Section 4.5. Recent work has extended causal read-only transactions to the client caches [39]. While client caches are indeed not full replicas, [39] continues to assume fully replicated causal stores.

TxCache is a programmer-visible, application data caching system that leverages programmer-specified staleness tolerance to enable caching with transactional consistency [30]. In contrast, *Karma*'s caches are transparent to applications and programmers. Unlike TxCache, *Karma* does not cache the results of application functions. As such, TxCache can be used atop *Karma* for such "computation caching".

8 Conclusion

We have presented *Karma*, the first partitioned, causally-consistent data store to support partial data replication with storage caching while offering consistency and availability under partition. *Karma*'s novel dynamic ring binding mechanism enables it to guarantee full availability under a single AZ failure, and simple network partition modes. Evaluations using a test-bed emulating geo-replicated settings show that *Karma* achieves 43% higher throughput on average and significantly lower read latencies than COPS-PR, while incurring similar costs. Further, *Karma*'s two key features – dynamic ring binding and caching – are important for significantly better performance under both normal conditions and network congestion than COPS-PR. Finally, *Karma* can reduce compute and storage costs by $32-60\%$, and bandwidth costs by $14-31\%$ compared to COPS-Ideal. Overall, our work is an important step forward towards making causally consistent cloud storage systems practical.

References

[1] Phillipe Ajoux, Nathan Bronson, Sanjeev Kumar, Wyatt Lloyd, and Kaushik Veeraraghavan. Challenges to adopting stronger consistency at scale. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. USENIX Association.

[2] Sérgio Almeida, Joao Leitao, and Luís Rodrigues. Chainreaction: a causal+ consistent datastore based on chain replication. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 85–98. ACM, 2013.

[3] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. The potential dangers of causal consistency and an explicit solution. In *Proceedings of the Third ACM Symposium on Cloud Computing (SOCC)*, page 22. ACM, 2012.

[4] Peter Bailis, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. Bolt-on causal consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 761–772. ACM, 2013.

[5] Jason Baker, Chris Bond, James Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Léon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, volume 11, pages 223–234, 2011.

[6] Nalini Moti Belaramani, Michael Dahlin, Lei Gao, Amol Nayate, Arun Venkataramani, Praveen Yalagandula, and Jiandan Zheng. PRACTI replication. In *NSDI*, volume 6, pages 5–5, 2006.

[7] Manuel Bravo, Luís Rodrigues, and Peter Van Roy. Towards a scalable, distributed metadata service for causal consistency under partial geo-replication. In *Proceedings of the Doctoral Symposium of the 16th International Middleware Conference*, Middleware Doct Symposium '15, pages 5:1–5:4, New York, NY, USA, 2015. ACM.

[8] E. Brewer. Cap twelve years later: How the "rules" have changed. *Computer*, 45(2):23–29, Feb 2012.

[9] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. Tao: Facebook's distributed data store for the social graph. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 49–60, San Jose, CA, 2013. USENIX.

[10] Marta Carbone and Luigi Rizzo. Dummynet revisited. *ACM SIGCOMM Computer Communication Review*, 40(2):12–20, 2010.

[11] Brian F Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. volume 1, pages 1277–1288. VLDB Endowment, 2008.

[12] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing (SOCC)*, pages 143–154, 2010.

[13] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *OSDI*, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association.

[14] Tyler Crain and Marc Shapiro. Designing a causally consistent protocol for geo-distributed partial replication. In *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '15, pages 6:1–6:4, New York, NY, USA, 2015. ACM.

[15] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *SOSP*, volume 7, pages 205–220, 2007.

[16] Jiaqing Du, Sameh Elnikety, Amitabha Roy, and Willy Zwaenepoel. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *2013 ACM Symposium on Cloud Computing (SOCC)*, number EPFL-CONF-188527, 2013.

[17] Jiaqing Du, Călin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. Gentlerain: Cheap and scalable causal consistency

with physical clocks. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC)*, pages 1–13. ACM, 2014.

[18] Garth Gibson, Gary Grider, Andree Jacobson, and Wyatt Lloyd. Probe: A thousand-node experimental cluster for computer systems research. In *USENIX ;login:*, volume 38, June 2013.

[19] David K. Gifford. Weighted voting for replicated data. In *SOSP*, 1979.

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

[21] Amazon EC2 Availability Zones. http://aws.amazon.com/ec2/faqs.

[22] Tim Kraska, Gene Pang, Michael J Franklin, Samuel Madden, and Alan Fekete. Mdcc: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 113–126. ACM, 2013.

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

[24] Leslie Lamport. Paxos Made Simple.

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

[26] Rui Li, Shengjie Wang, Hongbo Deng, Rui Wang, and Kevin Chen-Chuan Chang. Towards social user profiling: unified and discriminative influence model for inferring home locations. In *ACM SIGKDD*, pages 1023–1031. ACM, 2012.

[27] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *SOSP*, 2011.

[28] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Stronger semantics for low-latency geo-replicated storage. In *NSDI*. USENIX, 2013.

[29] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In *SOSP*, 1997.

[30] Dan RK Ports, Austin T Clements, Irene Zhang, Samuel Madden, and Barbara Liskov. Transactional consistency and automatic management in an application data cache. In *OSDI*, volume 10, pages 1–15, 2010.

[31] Amazon EC2 Regions and Availability Zones. http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html.

[32] Min Shen, Ajay D Kshemkalyani, and Ta-yuan Hsu. Opcam: Optimal algorithms implementing causal memories in shared memory systems. University of Illinois at Chicago, Dept. of Computer Science, 2014.

[33] Min Shen, Ajay D Kshemkalyani, and Ta-yuan Hsu. Causal consistency for geo-replicated cloud storage under partial replication. In *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*, pages 509–518. IEEE, 2015.

[34] Min Shen, Ajay D Kshemkalyani, and Ta-yuan Hsu. Opcam: Optimal algorithms implementing causal memories in shared memory systems. In *Proceedings of the 2015 International Conference on Distributed Computing and Networking*, page 16. ACM, 2015.

[35] Yair Sovran, Russell Power, Marcos K Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *SOSP*. ACM, 2011.

[36] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 309–324, New York, NY, USA, 2013. ACM.

[37] Robbert van Renesse and Fred B Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, volume 4, pages 91–104, 2004.

[38] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V Madhyastha. SPANStore: cost-effective geo-replicated storage spanning multiple cloud services. In *SOSP*. ACM, 2013.

[39] Marek Zawirski, Nuno Preguiça, Sérgio Duarte, Annette Bieniusa, Valter Balegas, and Marc Shapiro. Write fast, read in the past: Causal consistency for client-side applications. In *Proceedings of the 16th Annual Middleware Conference*, Middleware '15, pages 75–87, New York, NY, USA, 2015. ACM.

[40] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K Aguilera, and Jinyang Li. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *SOSP*. ACM, 2013.