Cache Serializability: Reducing Inconsistency in Edge Transactions

Ittay Eyal Ken Birman Robbert van Renesse
Cornell University

***Abstract*—Read-only caches are widely used in cloud infrastructures to reduce access latency and load on backend databases. Operators view coherent caches as impractical at genuinely large scale and many client-facing caches are updated asynchronously with best-effort pipelines. Existing solutions that support cache consistency are inapplicable to this scenario since they require a round trip to the database on every cache transaction.**

Existing incoherent cache technologies are oblivious to transactional data access, even if the backend database supports transactions. We propose *T-Cache*, a novel caching policy for read-only transactions in which inconsistency is tolerable (won't cause safety violations) but undesirable (has a cost). T-Cache improves cache consistency despite asynchronous and unreliable communication between the cache and the database. We define *cache-serializability*, a variant of serializability that is suitable for incoherent caches, and prove that with unbounded resources T-Cache implements this new specification. With limited resources, T-Cache allows the system manager to choose a trade-off between performance and consistency.

Our evaluation shows that T-Cache detects many inconsistencies with only nominal overhead. We use synthetic workloads to demonstrate the efficacy of T-Cache when data accesses are clustered and its adaptive reaction to workload changes. With workloads based on the real-world topologies, T-Cache detects $43-70\%$ of the inconsistencies and increases the rate of consistent transactions by $33-58\%$.

I. Introduction

Internet services like online retailers and social networks store important data sets in large distributed databases. Until recently, technical challenges have forced such large-system operators to forgo transactional consistency, providing per-object consistency instead. In contrast, backend systems often support transactions with guarantees such as snapshot isolation and even full transactional atomicity [1], [2], [3], [4].

Our work begins with the observation that it can be difficult for client-tier applications to leverage the transactions that the databases provide: transactional reads satisfied primarily from edge caches cannot guarantee coherency. Yet, by running from cache, client-tier transactions shield the backend database from excessive load, and because caches are typically placed close to the clients, response latency can be improved. The result is a tradeoff: we run transactions against incoherent caches, and each time a cache returns inconsistent data, the end-user is potentially exposed to visibly incorrect results.

The problem centers on the asynchronous style of communication between the database and the geo-distributed caches. A cache must minimize the frequency of backend interactions; any approach requiring a significant rate of round-trips to the database would incur unacceptable latency. A cache must also respond promptly, which rules out asynchronous update policies that might lock cache entries. The latency from cache to database and back is often high, ruling out cache coherency schemes that would require the backend database to invalidate or update cached objects. In many settings the backend can't even track the locations at which cached objects might reside. Here, we define a variant of serializability called cache-serializability that is suitable for incoherent caches.

Many web applications, from social networks to online retailers, run over caches oblivious to transactional consistency. Cache-level inconsistencies do not crash these systems, but the end-user may be frustrated and platform revenue is subsequently reduced. With T-Cache, the benefits of reading from an edge cache are maintained, but the frequency of these negative costs is reduced. Our protocol is especially beneficial for workloads where data accesses are clustered, which is common in today's large-scale systems. T-Cache achieves this by storing dependency information with the cached objects, allowing the cache to identify possible inconsistencies without contacting the database. The user can improve the level of consistency by adjusting the size of this dependency data: more dependency data leads to increased consistency.

To demonstrate the efficacy of the proposed scheme, we created a prototype implementation and exposed it to workloads based on graphically-structured real-world data, such as those seen in social-networking situations. The method detects $43-70\%$ of the inconsistencies and can increase the ratio of consistent transactions by $33-58\%$, both with low overhead. We construct synthetic workloads and explore the behavior of T-Cache with different degrees of data clustering, and also investigate its rate of response when clusters change.

With perfectly clustered workloads, T-Cache implements full cache-serializability. To explain this perfect behavior we prove a related claim — we show that with unbounded resources T-Cache implements cache-serializability.

In summary, the contributions of this work are:

1) Definition of cache-serializability, a variant of serializability suitable for incoherent caches.
2) T-Cache: An architecture that allows trading off efficiency and transaction-consistency in large cache deployments.
3) Evaluation of T-Cache with synthetic workloads, demonstrating its adaptivity and sensitivity to clustering.
4) Evaluation of T-Cache with workloads based on graphically-structured real-world data.
5) Proof that T-Cache with unbounded resources implements cache-serializability.

II. Motivation

a) Two-tier structure: Large Internet services store vast amounts of data. Online retailers such as Amazon and eBay maintain product stocks and information, and social networking sites such as Facebook and Twitter maintain graphical databases representing user relations and group structures. For throughput, durability, and availability, such databases are sharded and replicated.

The vast majority of accesses are read-only (e.g., Facebook reports a 99.8% read rate [5]). To reduce database load and to reduce access latency, these companies employ a two-tier structure, placing layers of cache servers in front of the database (see Figure 1).

The caches of primary interest to us are typically situated far from the backend database systems — to reduce latency, companies place caches close to clients. Timeouts are used to ensure that stale cached objects will eventually be flushed, but to achieve a high cache hit ratio, timeout values are generally large. To obtain reasonable consistency, the database sends an asynchronous stream of invalidation records or cache updates, often using protocols optimized for throughput and freshness and lacking absolute guarantees of order or reliability.

It is difficult to make this invalidation mechanism reliable without hampering database efficiency. First, the databases themselves are large, residing on many servers, and may be geo-replicated. Databases supporting cloud applications are often pipelined, using locks prudently in order to maximize concurrency but ensuring that there is always other work to do; the result is that some updates complete rapidly but others can exhibit surprisingly high latency. During these delays, read access must go forward. Databases cannot always accurately track the caches that hold a copy of each object, because the context makes it very hard to send timely invalidations: they could be delayed (e.g., due to buffering or retransmissions after message loss), not sent (e.g., due to an inaccurate list of locations), or even lost (e.g., due to a system configuration change, buffer saturation, or because of races between reads, updates, and invalidations). A missing invalidation obviously leaves the corresponding cache entry stale. Pitfalls of such invalidation schemes are described in detail by Nishita et al. [6] and by Bronson et al. [5].

b) DB Transactional consistency: The complexity of implementing geo-scale databases with strong guarantees initially led companies to abandon cross-object consistency altogether and make do with weak guarantees such as per-object atomicity. In effect, such systems do repair any problems that arise, eventually, but the end-user is sometimes exposed to inconsistency. For some applications this is acceptable, and the approach has been surprisingly successful. In today's cloud, relaxed consistency is something of a credo.

But forgoing transactional consistency can result in undesired behavior of a service. Consider a buyer at an online site who looks for a toy train with its matching tracks just as the vendor is adding them to the database. The client may see only the train in stock but not the tracks because the product insertion transaction would often be broken into two or more atomic but independent sub-transactions. In a social network, an inconsistency with unexpected results can occur if a user x's record says it belongs to a certain group, but that group's record does not include x. Web albums maintain picture data and access control lists (ACLs) and it is important that ACL and album updates are consistent (the classical example involves removing one's boss from the album ACL and then adding unflattering pictures).

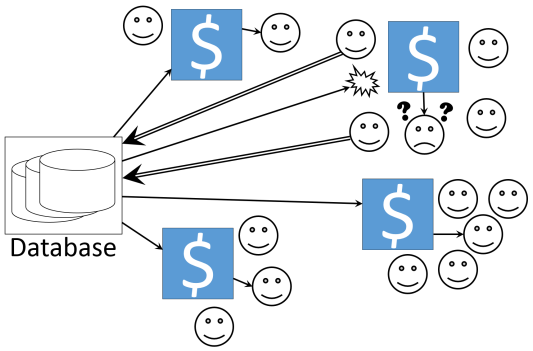


Fig. 1. The common two-tiered structure. Clients perform read-only transactions by accessing caches, which receive their values by reading from the database (solid lines). Update transactions go directly to the database (double lines). Subsequent cache invalidations can be delayed or even lost due to race conditions, potentially leading to inconsistent views by the clients.

As a result, we see more and more systems that are basically transactional in design and structure, but incorporate workarounds to tolerate weak consistency. When inconsistency does arise, the platform's utility is reduced. This is our target: our goal is to reduce the frequency of such events as much as possible. There has been a wave of recent innovations within the backend, offering scalable object stores that can efficiently support transactions through snapshot isolation and even full atomicity [1], [2], [3], [4]. Our approach seeks to improve transaction consistency at the cache layer, without requiring the cache to access the backend on each read or to invalidate items that may be undergoing updates.

c) Caches bring value but cause problems: As noted, many of today's systems are transactional at the backend, yet inconsistent because they use edge caching decoupled from the internal transactional mechanisms. Even when the database itself is consistent, the vast majority of operations are read-only transactions issued by edge clients and are at high risk of observing inconsistent state in the cache. The outright loss of cache invalidations emerges as an especially significant problem if transactional consistency is of value. In our work, we start by accepting that any solution must preserve performance properties of the existing caching tier. In particular, we need to maintain the shielding role of the cache: the cache hit ratio should be high. Additionally, a read-only cache access should complete with a single client-to-cache round-trip on cache hits. This rules out coherent cache solutions such as [7]. On the other hand, our approach gains leverage by adopting the view that inconsistency is undesirable but not unacceptable. Our goal is thus to *reduce the rate of user-visible inconsistencies.*

III. Architecture

Since the cache is required to respond immediately to the client on hits, and the database-cache channel is asynchronous, we decided to employ a transactional consistency that is weaker than the full ACID model. In our approach, read-only transactions and update transactions that access the same cache are guaranteed an atomic execution, but read-only transactions that access different caches may observe different orderings for independent update transactions.

Definition 1 (Cache serializability)**.** *For every execution σ, every partial execution that includes all update transactions in σ and all read-only transactions that go through a single cache server, is serializable.*

Our solution seeks to come as close as possible to cache serializability, subject to constraints imposed by the bounded size of the edge caches and the use of asynchronous communication with the DB. We start with an observation: in many scenarios, objects form *clusters* with strong locality properties. Transactions are likely to access objects that are, in some sense, close to each other. For retailers this might involve related products, for social networks the set of friends, for geographical services physical proximity, and for web albums the ACL objects and the pictures assigned to them. Moreover, in some cases applications explicitly cluster their data accesses to benefit from improved parallelism [8]. The resulting transactions access objects from a single cluster, although there will also be some frequency of transactions that access unrelated objects in different clusters.

Our solution requires minor changes to the database object representation format, imposing a small and constant memory overhead (that is, independent of the database size and the transaction rate). This overhead involves tracking and caching what we refer to as *dependency lists.* These are bounded-length lists of object identifiers and the associated version numbers, each representing some recently updated objects upon which the cached object depends.

A bounded-sized list can omit dependency information required to detect inconsistencies, hence it is important to use a bound large enough to capture most of the relevant dependencies. At present we lack an automated way to do this: we require the developer to tune the length so that the frequency of errors is reduced to an acceptable level, reasoning about the trade-off (size versus accuracy) in a manner we discuss further below. Intuitively, dependency lists should be roughly the same size as the size of the workload's clusters.

Our extensions offer a transactional interface to the cache in addition to the standard read/write API. In many cases, our algorithm detects and fixes inconsistent read-only transactions at the cache with constant complexity. It does so by either aborting the transaction (which can then be retried), or invalidating a cached object which can then force a read from the database (similar to handling cache misses). When the dependency lists fail to document a necessary dependency, an application might be exposed to stale values.

Because we have in mind client-side applications that are unlikely to validate against the back-end, for many of our intended uses some level of undetected inconsistency can slip past. However, because the developer would often be able to tune the mechanism, during steady-state operation of large applications, the rate of unnoticed inconsistencies could be extremely low.

With clustered workloads we will demonstrate that it is sufficient to store a small set of dependencies to detect most inconsistencies. We also investigate workloads where the clustered access pattern is less strongly evident; here, our approach is less effective even with longer dependency list lengths. Thus our solution is not a panacea, but, for applications matched to our assumptions, can be highly effective.

A. Database

We assume that the database tags each object with a version number specific to the transaction that most recently updated it, and that there is a total ordering on version numbers. The version of a transaction is chosen to be larger than the versions of all objects accessed by the transaction. The database stores for each object o a list of k dependencies $(d_1^o, v_1^o), (d_2^o, v_2^o), \ldots (d_k^o, v_k^o)$. This is a list of identifiers and versions of other objects that the current version of o depends on. A read-only transaction that sees the current version of o must not see object d_i with version smaller than v_i.

When a transaction t with version v_t touches objects o_1 and o_2, it updates both their versions and their dependency lists. Subsequent accesses to object o_1 must see object o_2 with a version not smaller than v_t. Moreover, it inherits all of the l dependencies of o_2 (where l is the length of o_2's dependency list). So the dependency list of o_1 becomes

$$(d_1^{o_1}, v_1^{o_1}), \ldots (d_k^{o_1}, v_k^{o_1}), (o_2, v_t), (d_2^{o_2}, v_2^{o_2}), \ldots (d_l^{o_2}, v_l^{o_2}) \ .$$

When a transaction is committed, this update is done for all objects in the transaction at once. Given a read set *readSet*, and a write set *writeSet*, containing tuples comprised of the keys accessed, their versions and their dependency lists, the database aggregates them to a single full dependency list:

$$\textit{full-dep-list} \leftarrow \bigcup_{\substack{(key,ver,depList)\in \\ readSet\cup writeSet}} \{(key, ver)\} \cup \textit{depList} \ .$$

This list is pruned to match the target size using LRU, and stored with each write-set object. A list entry can be discarded if the same entry's object appears in another entry with a larger version. Nevertheless, were their lengths not bounded, dependency lists could quickly grow to include all objects in the database.

B. Cache

In our scheme, the cache interacts with the database in essentially the same manner as for a consistency-unaware cache, performing single-entry reads (no locks, no transactions) and receiving invalidations as the database updates objects. Unlike consistency-unaware caches, the caches read from the database not only the object's value, but also its version and the dependency list.

To its clients, the extended cache exports a transactional read-only interface. Client read requests are extended with a transaction identifier and a last-op flag

`read(txnID, key, lastOp) .`

The transaction identifier `txnID` allows the cache to recognize reads belonging to the same transaction. The cache responds with either the value of the requested object, or with an abort if it detects an inconsistency between this read and any of the previous reads with the same transaction ID. We do not guarantee that inconsistencies will be detected. The `lastOp` allows the cache to garbage-collect its transaction record after responding to the last read operation of the transaction. The cache will treat subsequent accesses with the same transaction ID as new transactions.

To implement this interface, the cache maintains a record of each transaction with its read values, their versions, and their dependency lists. On a read of key_{curr}, the cache first obtains the requested entry from memory (cache hit), or database (cache miss). The entry includes the value, version ver_{curr} and dependency list $depList_{curr}$. The cache checks the currently read object against each of the previously read objects. If a previously read version v' is older than the version v expected by the current read's dependencies

$$\exists k, v, v' : v > v' \wedge (k, v) \in depList_{curr} \wedge (k, v') \in readSet \cup writeSet \ , \quad (1)$$

or the version of the current read v_{curr} is older than the version v expected by the dependencies of a previous read

$$\exists v : v > v_{curr} \wedge (key_{curr}, v) \in readSet \cup writeSet \ , \quad (2)$$

an inconsistency is detected. Otherwise the cache returns the read value to the client.

Upon detecting an inconsistency, the cache can take one of three paths:

1) `ABORT`: abort the current transaction. Compared to the other approaches, this has the benefit of affecting only the running transaction and limiting collateral damage.
2) `EVICT`: abort the current transaction *and* evict the violating (too-old) object from the cache. This approach guesses that future transactions are likely to abort because of this object.
3) `RETRY`: check which is the violating object. If it is the currently accessed object (Equation 2), treat this access as a miss and respond to it with a value read from the database. If the violating object was returned to the user in a previous read (Equation 1), evict the stale object and abort the transaction (as in `EVICT`).

C. Consistency

With unbounded resources, T-Cache detects all inconsistencies, as stated in the following theorem.

Theorem 1. *T-Cache with unbounded cache size and unbounded dependency lists implements cache-serializability.*

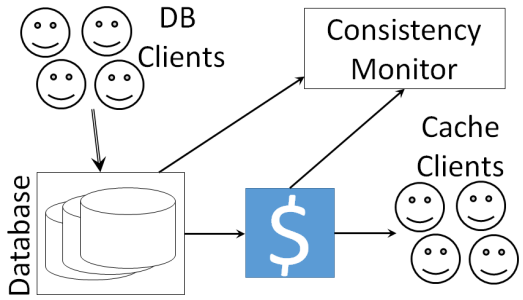


Fig. 2. Experimental setup. Update clients access database, which sends invalidations to the cache. Read-only clients access cache. Consistency monitor (experiment-only element) receives all transactions and rigorously detects inconsistencies for statistics.

The proof [9] is by constructing a serialization of the transactions in the database and in one cache, based on the fact that the transactions in the database itself are serializable.

The implications of Theorem 1 will be seen in Section V-A3. T-Cache converges to perfect detection when stable clusters are as large as its dependency lists. Then, its dependency lists are large enough to describe all dependencies.

IV. Experimental Setup

To evaluate the effectiveness of our scheme, we implemented a prototype. To study the properties of the cache, we only need a single column (shard) of the system, namely a single cache backed by a single database server.

Figure 2 illustrates the structure of our experimental setup. A single database implements a transactional key-value store with two-phase commit. A set of cache clients perform read-only transactions through a single cache server. The cache serves the requests from its local storage if possible, or reads from the database otherwise.

On startup, the cache registers an upcall that can be used by the database to report invalidations; after each update transaction, the database asynchronously sends invalidations to the cache for all objects that were modified. A ratio of 20% of the invalidations, chosen uniformly at random, are dropped by the experiment; this is extreme and would only be seen in the real world under conditions of overload or when the system configuration is changed.

Both the database and the cache report all completed transactions to a consistency monitor, created in order to gather statistics for our evaluation. This server collects both committed and aborted transactions and it maintains the full dependency graph. It performs full serialization graph testing [10] and calculates the rate of inconsistent transactions that committed and the rate of consistent transactions that were unnecessarily aborted.

Our prototype does not address the issue of cache eviction when running out of memory. In our experiments, all objects in the workload fit in the cache, and eviction is only done if there is a direct reason, as explained below. Had we modeled them, evictions would reduce the cache hit rate, but could not cause new inconsistencies.

We evaluate the effectiveness of our transactional cache using various workloads and varying the size of the dependency lists maintained by the cache and the database. For the cases considered, short dependency lists suffice (up to 5 versions per object). An open question for further study is whether there are workloads that might require limited but larger values. Note that dependencies arise from the topology of the object graph, and not from the size of the transactions' read and write sets.

As a baseline for comparison, we also implemented a timeout-based approach: it reduces the probability of inconsistency by limiting the life span of cache entries. We compare this method against our transactional cache by measuring its effectiveness with a varying time-to-live (TTL) for cache entries.

In all runs, both read and update transactions access 5 objects per transaction. Update clients access the database at a rate of 100 transactions per second, and read-only clients access the cache at a rate of 500 transactions per second.

Our experiment satisfies all read-only transactions from the cache, while passing all update transactions directly to the backend database. Each cache server is unaware of the other servers — it has its own clients and communicates directly with the backend database. The percentage of read-only transactions can be arbitrarily high or low in this situation: with more caches, we can push the percentage up. Our simulation focuses on just a single cache—it would behave the same had there been many cache servers.

V. Evaluation

T-Cache can be used with any transactional backend and any transactional workload. Performance for read-only transactions will be similar to non-transactional cache access: the underlying database is only accessed on cache misses. However, inconsistencies may be observed.

First, we will use synthetic workloads so we can evaluate how much inconsistency can be observed as a function of the amount of clustering in the workload. This also allows us to look at the dynamic behavior of the system, when the amount of clustering and the clustering formation change over time.

Next, we will look at workloads based on Amazon's product co-purchasing and Orkut's social network to see how much inconsistency T-Cache can detect as a function of dependency list length, and compare this with a TTL-based approach. We are also interested in overhead, particularly the additional load on the backend database that could form if the the rate of cache misses increases.

Section III-B presented three strategies for responding to inconsistency detection. For both the synthetic and realistic workloads, we compare the efficacy of the three strategies.

A. Synthetic Workloads

Synthetic workloads allow us to understand the efficacy of T-Cache as a function of clustering. For the experiments described here, we configured T-Cache with a maximum of 5 elements per dependency list.

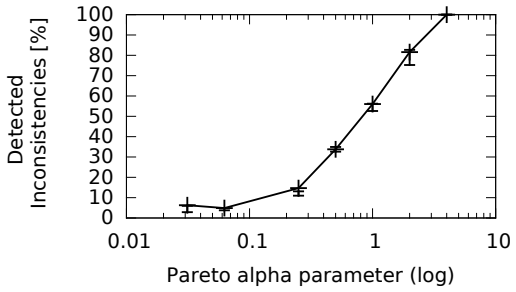


Fig. 3. Ratio of inconsistencies as a function of α.

Section V-A1 describes synthetic workload generation. Section V-A2 measures how many inconsistencies we can detect as a function of clustering and Section V-A3 considers clustering changes over time. Section V-A4 compares the efficacy of various approaches to dealing with detected inconsistencies.

1) Synthetic Workload Generation: Our basic synthetic workload is constructed as follows. We use 2000 objects numbered 0 through 1999. The objects are divided into clusters of size 5: $0-4, 5-9, 10-14, \ldots$, and there are two types of workloads. In the first, clustering is perfect and each transaction chooses a single cluster and chooses 5 times with repetitions within this cluster to establish its access set. In the second type of workloads access is not fully contained within each cluster. When a transaction starts, it chooses a cluster uniformly at random, and then picks 5 objects as follows. Each object is chosen using a bounded Pareto distribution starting at the head of its cluster i (a product of 5). If the Pareto variable plus the offset results in a number outside the range (i.e., larger than 1999), the count wraps back to 0 through $i-1$.

2) Inconsistency Detection as a Function of α: We start by exploring the importance of the cluster structure by varying the α parameter of the Pareto distribution. We vary the Pareto α parameter from 1/32 to 4. In this experiment we are only interested in detection, so we choose the ABORT strategy.

Figure 3 shows the ratio of inconsistencies detected by T-Cache compared to the total number of potential inconsistencies. At $\alpha = 1/32$, the distribution is almost uniform across the object set, and the inconsistency detection ratio is low — the dependency lists are too small to hold all relevant information. At the other extreme, when $\alpha = 4$, the distribution is so spiked that almost all accesses of a transaction are within a cluster, allowing for perfect inconsistency detection. We note that the rate of detected inconsistencies is so high at this point that much of the load goes to the backend database and saturates it, reducing the overall throughput.

3) Convergence: So far we have considered behavior with static clusters, that is, over the entire run of each experiment accesses are confined to the same (approximate) clusters. Arguably, in a real system, clusters change slowly, and so if T-Cache converges to maintain the correct dependency lists as clusters change, our setup serves as a valid quasi-static analysis.

In this section, we investigate the convergence of T-Cache when clusters change over time. Since the dependency lists

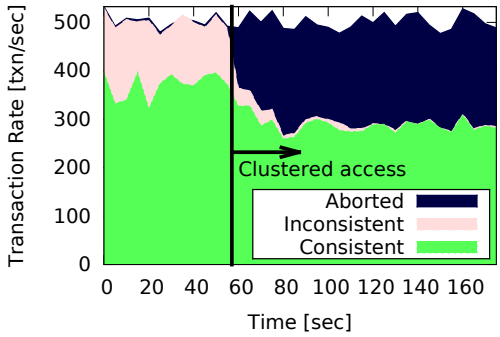


Fig. 4. Convergence of T-Cache. Before time $t = 58s$ accesses are uniformly at random. Afterward, accesses are clustered.

of the objects are updated using LRU, the dependency list of an object o tends to include those objects that are frequently accessed together with o. Dependencies in a new cluster automatically push out dependencies that are now outside the cluster.

Cluster formation: To observe convergence, we perform an experiment where accesses suddenly become clustered. Initially accesses are uniformly at random from the entire set (i.e., no clustering whatsoever), then at a single moment they become perfectly clustered into clusters of size 5. Transactions are aborted on detecting an inconsistency. We use a transaction rate of approximately 500 per second. The database includes 1000 objects.

Figure 4 shows the percentage of transactions that commit and are consistent (at the bottom), the percentage of transactions that commit but are inconsistent (in the middle), and the percentage of transactions that abort (at the top). Before $t = 58s$ access is unclustered, and as a result the dependency lists are useless; only few inconsistencies are detected, that is, about 26% of the transactions that commit have witnessed inconsistent data. At $t = 58s$, accesses become perfectly clustered. As desired, we see fast improvement of inconsistency detection. The inconsistency rate drops as the abort rate rises — this is desired as well. The overall rate of consistent committed transactions drops because the probability of conflicts in the clustered scenario is higher.

Drifting Clusters: To illustrate more realistic behavior, we use clustered accesses that slowly drift. Transactions are perfectly clustered, as in the previous experiment, but every 3 minutes the cluster structure shifts by 1 ($0-4, 5-9, 10-14 \rightarrow 1-4, 5-10, 11-15$, and wrapping back to zero after 1999). Figure 5 shows the results. After each shift, the objects' dependency lists are outdated. This leads to a sudden increased inconsistency rate that converges back to zero, until this convergence is interrupted by the next shift.

4) Detection vs. Prevention: Section III-B presented three possible strategies for the cache to deal with inconsistency detection: (1) aborting the transaction (ABORT), (2) aborting and evicting value (EVICT), and (3) read-through when possible as in cache miss, abort otherwise (RETRY). We will now compare their efficacies.

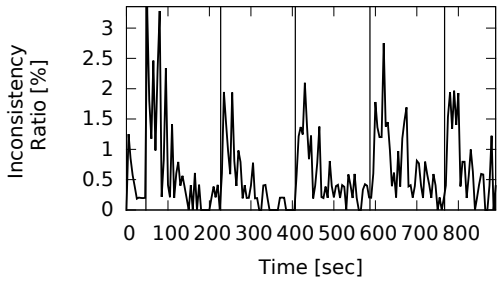


Fig. 5. Perfectly clustered synthetic workload where the clusters shift by 1 every 3 minutes, marked by vertical lines.

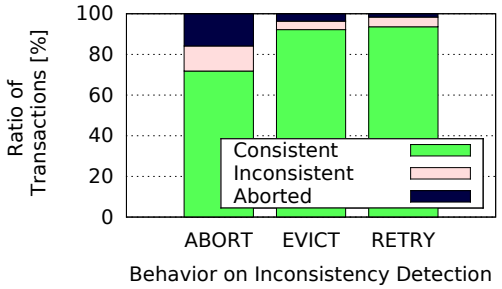


Fig. 6. The efficacy of T-Cache as a function of the strategy taken for handling detected inconsistencies. ABORT detects and aborts transactions that glimpsed inconsistent data. EVICT aborts, but also evicts the inconsistent cached data, and RETRY performs a read-through if it resolves an inconsistent read. In each case we break down the transactions as consistent, undetected inconsistencies, and aborted.

We use the approximate clusters workload with 2000 objects, a window size of 5, a Pareto α parameter of 1.0, and the maximum dependency list size is set to 5.

Figure 6 illustrates the results. For each strategy, the lower portion of the graph is the ratio of committed transactions that are consistent, the middle portion is committed transactions that are inconsistent, and the top portion is aborted transactions.

The abort strategy provides a significant improvement over a normal, consistency-unaware cache, as the strategy detects and aborts over 55% of all inconsistent transactions that would have been committed. But the other strategies make further improvements. EVICT reduces uncommittable transactions to 28% of its value with ABORT. This indicates that violating (too-old) cache entries are likely to be repeat offenders: they are too old for objects that are likely to be accessed together with them in future transactions, and so it is better to evict them. RETRY reduces uncommittable transactions further to about 23% of its value with ABORT.

B. Realistic Workloads

We now evaluate the efficacy of T-Cache with workloads based on two sampled topologies from the online retailer Amazon and the social network Orkut. Section V-B1 describes how we generated these workloads. Section V-B2 measures the efficacy of T-Cache on these workloads as a function of maximum dependency list size, and compares this to a strategy

based on TTLs. Section V-B3 compares the efficacy of the three strategies of dealing with detected inconsistencies.

1) Workload Generation: We generated two workloads based on real data:

1) Amazon: We started from a snapshot of Amazon's product co-purchasing graph taken early 2003 [11]. Each product sold by the online retailer is a node and each pair of products purchased in a single user session is an edge. The original graph contains more than 260,000 nodes.
2) Orkut: For the second, we used a snapshot of the friendship relations graph in the Orkut social network, taken late 2006 [12]. In this graph, each user is a node and each pair of users with a friend relationship is an edge. The original graph contains more than 3,000,000 nodes.

Because the sampled topologies are large and we only need to simulate a single "column" of the system for our purposes — one database server and one cache server — we down-sample both graphs to 1000 nodes. We use a technique based on random walks that maintains important properties of the original graph [13], specifically clustering which is central to our experiment. We start by choosing a node uniformly and random and start a random walk from that location. In every step, with probability 15%, the walk reverts back to the first node and start again. This is repeated until the target number of nodes have been visited. Figure 7(a) and (b) show a further down-sampling to 500 nodes to provide some perception of the topologies. The graphs are visibly clustered, the Amazon topology more so than the Orkut one, yet well-connected.

Treating nodes of the graphs as database objects, transactions are likely to access objects that are topologically close to one another. For the online retailer, it is likely that objects bought together are also viewed and updated together (e.g., viewing and buying a toy train and matching rails). For the social network, it is likely that data of befriended users are viewed and updated together (e.g., tagging a person in a picture, commenting on a post by a friend's friend, or viewing one's neighborhood).

Therefore, we generate a transactional workload that accesses products that are topologically close. Again, we use random walks. Each transaction starts by picking a node uniformly at random and takes 5 steps of a random walk. The nodes visited by the random walk are the objects the transaction accesses. Update transactions first read all objects from the database, and then update all objects at the database. Read transactions read the objects directly from the cache.

2) Efficacy and Overhead: We evaluate T-Cache using the workloads described above. We found that the abort rate is negligible in all runs. Efficacy is therefore defined to be the ratio of inconsistent transactions out of all commits.

The overhead of the system is twofold. First, dependency list maintenance implies storage and bandwidth overhead at both the database and the cache, as well as compute overhead for dependency list merging at the server and consistency checks at the cache. However, the storage required is only for object IDs and versions, not content, and both updates and checks are $O(1)$ in the number of objects in the system and $O(k^2)$ in the size of the dependency lists, which is limited to 5 in our experiments.

The second and potentially more significant overhead is the effect on cache hit ratio due to evictions and hence the database load. Since cache load is significantly larger than database load (2 orders of magnitude for Facebook [5]), even a minor deterioration in hit ratio can yield a prohibitive load on the backend database. Figure 7c shows the experiment results. Each data point is the result of a single run.

We vary the dependency list size and for each value run the experiment for the two workloads and measure the average values of these metrics. T-Cache is able to reduce inconsistencies significantly. For the retailer workload, a single dependency reduces inconsistencies to 56% of their original value, two dependencies reduce inconsistencies to 11% of their original value, and three to less than 7%. For the social network workload, with 3 dependencies fewer than 7% of the inconsistencies remain.

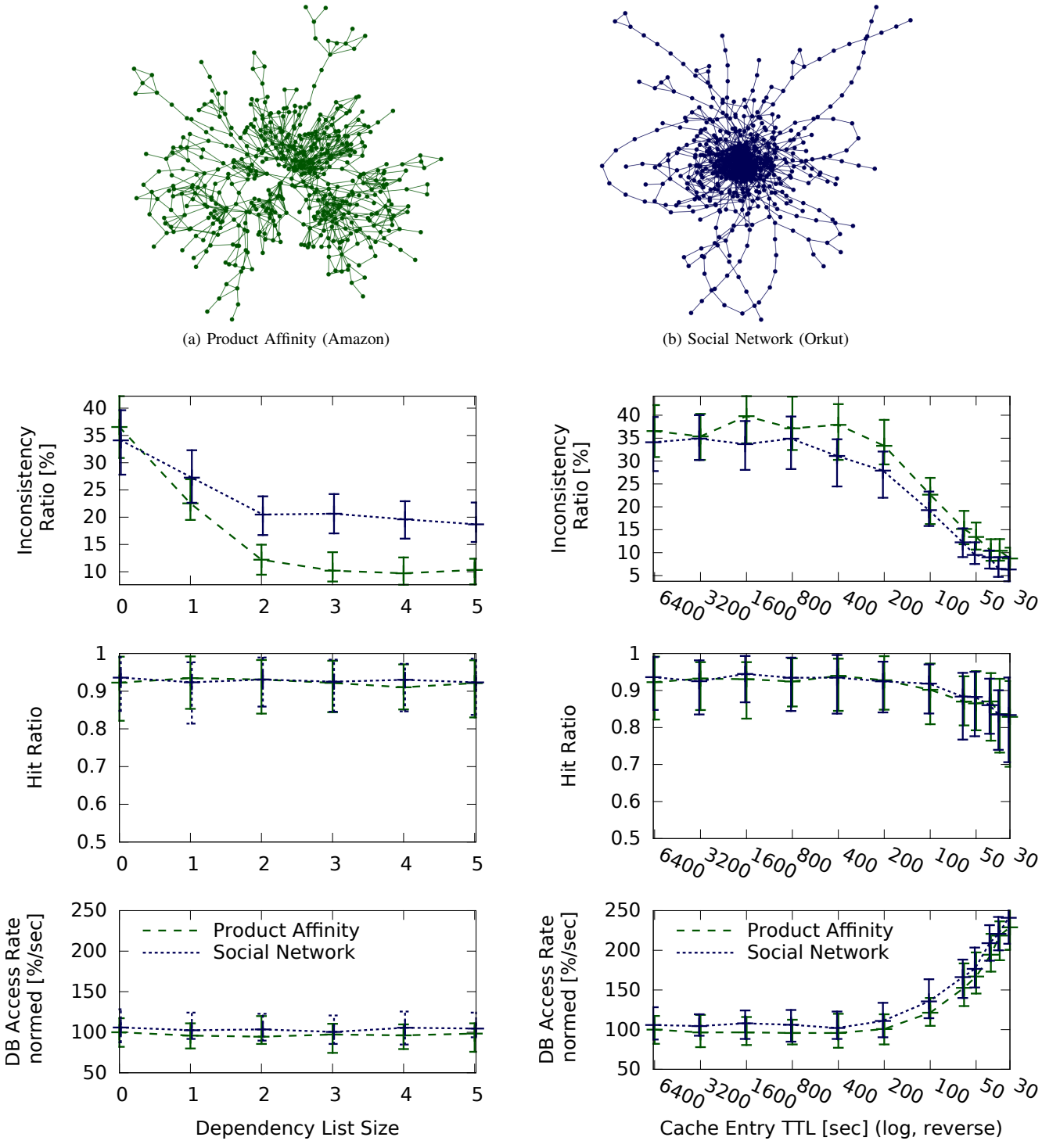
In both workloads there is no visible effect on cache hit ratio, and hence no increased access rate at the database. The reduction in inconsistency ratio is significantly better for the retailer workload. Its topology has a more clustered structure, and so the dependency lists hold more relevant information.

Next we compared our technique with a simple approach in which we limited the life span (Time To Live, TTL) of cache entries. Here inconsistencies are not detected, but their probability of being witnessed is reduced by having the cache evict entries after a certain period even if the database did not indicate they are invalid.

We run a set of experiments similar to the T-Cache ones, varying cache entry TTL to evaluate the efficacy of this method in reducing inconsistencies and the corresponding overhead. Compared to T-Cache, Limiting TTL has detrimental effects on cache hit ratio, quickly increasing the database workload. By increasing database access rate to more than twice its original load we only observe a reduction of inconsistencies of about 10%. This is more than twice the rate of inconsistencies achieved by T-Cache for the retailer workload and only slightly better than the rate of inconsistencies achieved by T-Cache for the social network workload.

3) Detection vs. Prevention: Figure 8 compares the efficacy of the ABORT, EVICT and RETRY policies with the Amazon and Orkut workloads. In these experiments we use dependency lists of length 3. Just as with the synthetic workload, evicting conflicting transactions is an effective way of invalidating stale objects that might cause problems for future transactions.

The effects are more pronounced for the well-clustered Amazon workload. With the Amazon workload, ABORT is able to detect 70% of the inconsistent transactions, whereas with the less-clustered Orkut workload it only detects 43%. In both cases EVICT reduces uncommittable transactions considerably, relative to their value with ABORT — 20% with the Amazon workload and 36% with Orkut. In the Amazon workload, RETRY further reduces this value to 11% of its value with ABORT.



(a) Product Affinity (Amazon)

(b) Social Network (Orkut)

(c) Transactional Cache

(d) Limited Cache Entry TTL

Fig. 7. Experiments with workloads based on a web retailer product affinity topology and a social network topology illustrated in (a) and (b). Transactional cache (c) compared against the alternative of reducing cache entry time-to-live (d). Data points are medians and error bars bound the 10 and 90 percentiles.

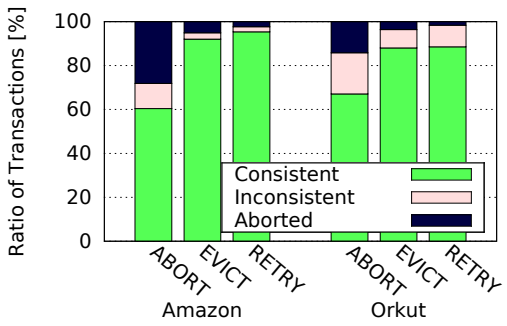


Fig. 8. The efficacy of T-Cache as a function of the inconsistency handling strategy for realistic workloads.

VI. Related Work

a) Scalable Consistent Databases: Recent years have seen a surge of progress in the development of scalable object stores that support transactions. Some systems such as [14], [15], [16], [17], [18] export novel consistency definitions that allow for effective optimizations. Several recent systems implement full fledged atomicity while preserving the system's scalability with a wide variety of workloads. Google's Spanner [1] utilizes accurate clock synchronization. Tango [2] by Balakrishnan et al. is constructed on top of the scalable Corfu [19] log. Eyal et al. [3] utilize a large set of independent logs. Escriva et al. [4] use DHT-based locking. Zhang et al. [20] use lock chains and assume transactions are known in advance. These methods scale well and often allow databases to accept loads similar to those handled by non-transactional databases. Nevertheless, they are not expected to disrupt the prevailing two-tier structure; *caches remain invaluable*.

Note that we are addressing the problem of read-only incoherent caches that respond to queries without access to the backend database. Previous work on coherent caches, e.g. [21], [22], [23], supports transactions using locks or communication with the database on each transaction. These techniques are not applicable in our scenario.

b) Consistent Caching: Much work has been done on creating consistent caches for web servers [24], [25], [26], [27], [28], distributed file systems [29], [30], Key-Value Stores [31], [5], [6] and higher level objects [32], [33]. Such systems consider only one object at a time, and only individual read and write operations, as they do not support a transactional interface. There are few if any multi-object or multi-operation consistency considerations. Such systems try to avoid staleness through techniques such as Time-To-Live (TTL), invalidation broadcasts, and leases. *Our work considers multi-object transactional consistency of cache access.*

c) Transactional Caching: Early work on scalable database caching mostly ignored transactional consistency [34]. Since then, work has been done on creating consistent caches for databases. TxCache [7] extends a centralized database with support for caches that provide snapshot isolation semantics, albeit the snapshots seen may be stale. To improve the commit rate for read-only transactions, they use *multiversioning*, where the cache holds several versions of an object and enables the cache to choose a version that allows a transaction to commit. This technique could also be used with our solution. Perez-Sorrosal et al. [35], [36] also support snapshot isolation, but can be used with any backend database, including ones that are sharded and/or replicated. JBossCache [37] provides a transactionally consistent cache for the JBoss middleware. Both JBossCache and [38] support transactions on cached Enterprise JavaBeans. [39] allows update transactions to read stale data out of caches and provide bounds on how much staleness is allowed. These techniques require fast communication between the cache and the database for good performance. In contrast, *in our work caches are asynchronously updated (or invalidated)*, which is how caches currently work in large multi-regional clouds.

VII. Conclusion

Existing large-scale computing frameworks make heavy use of edge caches to reduce client latency, but this form of caching has not been available for transactional applications. We believe this is one reason that transactions are generally not considered to be a viable option in extremely large systems.

We defined cache-serializability, a variant of serializability that is suitable for incoherent caches, which cannot communicate with the backend database on every read access. We then presented T-Cache, an architecture for controlling transaction consistency with caches. The system extends the edge cache by allowing it to offer a transactional interface. We believe that T-Cache is the first transaction-aware caching architecture in which caches are updated asynchronously. In particular, a lookup request only requires a round-trip to the database in case there is a cache miss — there is no additional traffic and delays to ensure cache coherence.

T-Cache associates dependency information with cached database objects, while leaving the interaction between the backend systems and the cache otherwise unchanged. This information includes version identifiers and bounded-length dependency lists. With even a modest amount of additional information, we show that inconsistency can be greatly reduced or even completely eliminated in some cases. With unbounded resources, we proved that T-Cache's algorithm implements cache-serializability.

T-Cache is intended for clustered workloads, which are common in social networks, product relationships, mobile applications with spatial locality, and many other cloud computing applications. Our experiments demonstrate T-Cache to be extremely effective in realistic workloads based on datasets from Amazon and Orkut. Using dependency lists of size 3, T-Cache detected $43-70\%$ of the inconsistencies, and was also able to increase consistent transaction rate by $33-58\%$ with only nominal overhead on the database. Our experiments with synthetic workloads showed that T-Cache's efficacy depends on the clustering level of the workload. T-Cache also responds well to change; we have shown that the system quickly adapts in workloads where data clustering evolves over time.

In future work, it may be possible to improve the performance of T-Cache by dynamically changing per-object dependency lists, and by allowing the application to explicitly inform the database and cache which dependencies are important.

Acknowledgement: The authors thank Dmitri Perelman for a conversation that inspired this work.

REFERENCES

[1] 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," *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, p. 8, 2013.

[2] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck, "Tango: Distributed data structures over a shared log," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 325–340.

[3] I. Eyal, K. Birman, I. Keidar, and R. van Renesse, "Ordering transactions with prediction in distributed object stores," in *Proc. of the 7th Workshop on Large-Scale Distributed Systems and Middleware (LADIS'13)*, 2013.

[4] R. Escriva, B. Wong, and E. G. Sirer, "Warp: Multi-key transactions for key-value stores," Dept. of Computer Science, Cornell University, Tech. Rep., 2013.

[5] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani, "TAO: Facebook's distributed data store for the social graph." in *USENIX Annual Technical Conference*, 2013, pp. 49–60.

[6] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling Memcache at Facebook," in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*. Lombard, IL: USENIX, 2013, pp. 385–398. [Online]. Available: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala

[7] D. R. Ports, A. T. Clements, I. Zhang, S. Madden, and B. Liskov, "Transactional consistency and automatic management in an application data cache." in *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, vol. 10, 2010, pp. 1–15.

[8] W. Xie, G. Wang, D. Bindel, A. Demers, and J. Gehrke, "Fast iterative graph computation with block updates," in *Proc. of the VLDB Endowment (PVLDB)*, vol. 6, Sep. 2013, pp. 2014–2025.

[9] I. Eyal, K. Birman, and R. van Renesse, "Controlled transactional consistency for web caching," *CoRR*, vol. abs/1409.8324, 2014. [Online]. Available: http://arxiv.org/abs/1409.8324

[10] P. A. Bernstein, *Concurrency control and recovery in database systems*. New York: Addison-Wesley, 1987, vol. 370.

[11] J. Leskovec, L. A. Adamic, and B. A. Huberman, "The dynamics of viral marketing," *ACM Transactions on the Web (TWEB)*, vol. 1, no. 1, p. 5, 2007.

[12] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and analysis of online social networks," in *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement (IMC'07)*. ACM, 2007, pp. 29–42.

[13] J. Leskovec and C. Faloutsos, "Sampling from large graphs," in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2006, pp. 631–636.

[14] 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 *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP 11)*, Oct. 2011.

[15] Y. Sovran, R. Power, M. K. Aguilera, and J. Li, "Transactional storage for geo-replicated systems," in *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP 11)*, Oct. 2011.

[16] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguica, and R. Rodrigues, "Making geo-replicated systems fast as possible, consistent when necessary," in *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*, 2012.

[17] P. Bailis, A. Fekete, J. M. Hellerstein, A. Ghodsi, and I. Stoica, "Scalable atomic visibility with ramp transactions," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14. New York, NY, USA: ACM, 2014, pp. 27–38. [Online]. Available: http://doi.acm.org/10.1145/2588555.2588562

[18] C. Xie, C. Su, M. Kapritsos, Y. Wang, N. Yaghmazadeh, L. Alvisi, and P. Mahajan, "Salt: Combining ACID and BASE in a distributed database," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, 2014.

[19] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobber, M. Wei, and J. D. Davis, "Corfu: A shared log design for flash clusters." in *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI'12)*, 2012, pp. 1–14.

[20] 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*. ACM, 2013, pp. 276–291.

[21] M. J. Franklin, M. J. Carey, and M. Livny, "Transactional client-server cache consistency: alternatives and performance," *ACM Transactions on Database Systems (TODS)*, vol. 22, no. 3, pp. 315–363, 1997.

[22] M. J. Carey, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, and M. J. Zwilling, "Shoring up persistent applications," in *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '94. New York, NY, USA: ACM, 1994, pp. 383–394. [Online]. Available: http://doi.acm.org/10.1145/191839.191915

[23] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari, "Efficient optimistic concurrency control using loosely synchronized clocks," *ACM SIGMOD Record*, vol. 24, no. 2, pp. 23–34, 1995.

[24] J. Challenger, A. Iyengar, and P. Dantzig, "A scalable system for consistently caching dynamic web data," in *Proc. INFOCOM '99*, Mar. 1999.

[25] H. Yu, L. Breslau, and S. Shenker, "A scalable web cache consistency architecture," *SIGCOMM Computer Communications Review*, vol. 29, no. 4, pp. 163–174, 1999.

[26] H. Zhu and T. Yang, "Class-based cache management for dynamic web content," in *Proc. INFOCOM '01*, 2001.

[27] M. Attar and M. Ozsu, "Alternative architectures and protocols for providing strong consistency in dynamic web applications," *World Wide Web Journal*, vol. 9, no. 3, pp. 215–251, 2006.

[28] Oracle, "Oracle web cache," http://www.oracle.com/technetwork/-middleware/webtier/overview.

[29] C. A. Kent, "Cache coherence in distributed systems," Ph.D. dissertation, Purdue University, Aug. 1986.

[30] A. M. Vahdat, P. C. Eastham, and T. E. Anderson, "Webfs: A global cache coherent file system," UC Berkeley, Tech. Rep., Dec. 1996.

[31] Memcached, "Memcached: a distributed memory object caching system," http://memcached.org.

[32] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler, "Scalable, distributed data structures for internet service construction," in *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation (OSDI'00)*, vol. 4. USENIX Association, 2000.

[33] R. Bakalova, A. Chow, C. Fricano, P. Jain, N. Kodali, D. Poirier, S. Sankaran, and D. Shupp, "WebSphere dynamic cache: Improving J2EE application experience," *IBM Systems Journal*, vol. 43, no. 2, 2004.

[34] Q. Luo, S. Krishnamurthy, C. Mohan, H. Pirahesh, H. Woo, B. G. Lindsay, and J. F. Naughton, "Middle-tier database caching for e-business," in *International Conference on Management of Data (SIGMOD)*, 2002, pp. 600–611.

[35] F. Perez-Sorrosal, M. Patino-Martinez, R. Jimenez-Peris, and B. Kemme, "Consistent and scalable cache replication for multi-tier J2EE applications," in *Proc. of Middleware'07*, 2007.

[36] F. Perez-Sorrosal, M. Patiño-Martinez, R. Jimenez-Peris, and B. Kemme, "Elastic SI-Cache: consistent and scalable caching in multi-tier architectures," *The International Journal on Very Large Data Bases (VLDB Journal)*, vol. 20, no. 6, pp. 841–865, 2011.

[37] M. Surtani and B. Ban, "JBoss Cache," http://jbosscache.jboss.com.

[38] A. Leff and J. Rayfield, "Improving application throughput with enterprise JavaBeans caching," in *International Conference on Distributed Computing Systems (ICDCS)*, 2003.

[39] P. A. Bernstein, A. Fekete, H. Guo, R. Ramakrishnan, and P. Tamma, "Relaxed-currency serializability for middle-tier caching and replication," in *International Conference on Management of Data (SIGMOD)*, 2006, pp. 599–610.