Rejig: A Scalable Online Algorithm for Cache Server Configuration Changes

Shahram Ghandeharizadeh(✉), Marwan Almaymoni, and Haoyu Huang

Computer Science Department, USC, Los Angeles, CA 90089-0781, USA
{shahram,almaymon,haoyuhua}@usc.edu

Abstract. A cache server configuration describes an assignment of data fragments to cache manager instances (CMIs). A load balancer may change this assignment by migrating fragments from one CMI to another. Similarly, an auto-scaling component may change the assignment by either inserting or removing CMIs in response to load fluctuations. These changes may generate stale cache entries. Rejig is a scalable online algorithm that manages configuration changes while providing read-after-write consistency. It is novel for several reasons. First, it allows for a subset of its clients and CMIs to use different configurations. Second, its client components propagate configuration changes to one another on demand and by using CMIs. This enables Rejig to scale and support diverse application classes including trusted mobile clients accessing the caching layer. When clients have intermittent network connectivity, Rejig detects if their cached configurations may result in stale data and updates them to the latest with no performance impact on either the CMIs or other clients. Rejig's overhead is in the form of 4 extra bytes of memory per cache entry and 4 extra bytes of the network bandwidth per request from a client to a CMI.

1 Introduction

Caches such as memcached [32], Redis [35], Ignite [15], KOSAR [17], and others improve the performance of traditional database management systems with workloads that exhibit a high read to write ratio [6,7,40]. A caching layer may consist of tens of servers for a small installation and thousands of servers with a popular site such as Facebook [33].

A physical server with many cores may host several Cache Manager Instances, CMIs. Each CMI is a process that might be multi-threaded. It is assigned a fixed number of cores and some amount of memory. It is also assigned a fraction of cache entries, a *fragment*. Multiple fragments are assigned to one CMI for load balancing. A load balancer may consider factors such as imposed load and cache hit rate to adjust the assignment of fragments to CMIs to enhance a performance metric such as system throughput [1,38].

A *configuration* is an assignment of fragments to CMIs. A coordinator manages configuration changes. A configuration changes due to: (1) addition or removal of CMIs by an auto-scaling component (or a system administrator),

A. Hameurlain and R. Wagner (Eds.): TLDKS XLII, LNCS 11860, pp. 111–134, 2019.
https://doi.org/10.1007/978-3-662-60531-8_5

(2) re-assignment of fragments to CMIs by a load-balancer, (3) re-assignment of fragments to CMIs in the presence of network partitions, (4) re-partitioning of data across fragments by a re-organization component in the form of either increasing or decreasing the number of fragments, or (5) a combination of these.

Configuration changes must preserve the application's read-after-write consistency defined as a read of a cache entry observing the value produced by the last committed write of the entry [31]. Configuration changes may compromise read-after-write consistency for two reasons. First, during the window of time when the coordinator publishes a new configuration, a few clients may have the old configuration while others have the new configuration. This discrepancy may cause two or more clients that reference the same cache entry to contact different CMIs, observing different values. If they write this cache entry then they will generate different values in different CMIs. The value observed by a subsequent read depends on whether this read is issued using the old or the new configuration, potentially compromising read-after-write consistency and correctness of an application. Second, a configuration change may re-assign a fragment to a new destination CMI without physically deleting its cache entries from the source CMI, leaving these entries to become cold at the source CMI and evicted by its cache replacement technique. In the presence of updates and a subsequent configuration change that assigns the fragment back to the source, the application may observe stale cache entries. To elaborate, consider a system that migrates F_k from CMI_i to CMI_j without deleting F_k's cache entries stored at CMI_i. Should the application update F_k's cache entries assigned to CMI_j then the value of their replicas on CMI_i become stale. If a subsequent configuration change assigns F_k back to CMI_i, references to these cold cache entries observe stale values. This violates read-after-write consistency.

An ideal solution to these two challenges should (1) be **pauseless** by processing user requests in the presence of configuration changes, (2) provide **read-after-write consistency** that guarantees data produced by a committed write is observed by all subsequent reads, (3) be **agnostic to the number of clients**, and (4) preserve as many **valid keys** as possible in the presence of configuration changes. With the latter, with v fragments assigned to CMI_i, if a configuration change assigns one fragment to a different CMI then keys of the remaining $v-1$ fragments of CMI_i should remain valid.

The primary *contribution* of this paper is Rejig[1] [16], a scalable online algorithm that satisfies the above requirements. Rejig extends existing systems [21,33]. While it allows multiple clients to use different configurations, it guarantees consensus on a fragment's replica by requiring clients that reference the fragment to use its latest CMI assignment always. Moreover, it allows cache entries of an old replica of a re-assigned fragment to become cold and evicted by the CMI's replacement technique. With those entries that remain, Rejig detects when the application references them and treats them as cache misses. Rejig may be configured to delete these cache entries to free the CMI's memory.

[1] The definition of Rejig is to rearrange or organize differently.

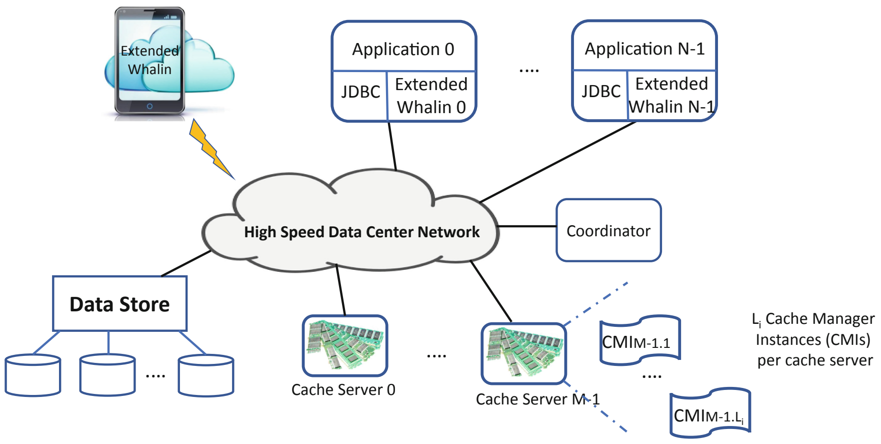
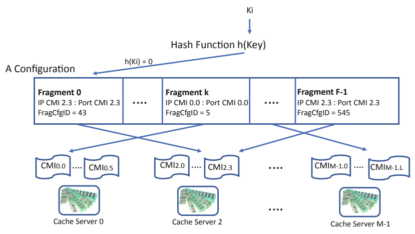
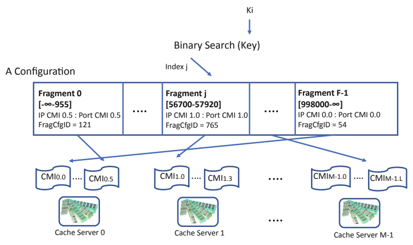


Fig. 1. A candidate architecture for Rejig. Concepts underlying Rejig are applicable to those architectures that tightly integrate the cache with the application [17] or represent the caching layer as a middleware [15, 28, 34].

Rejig is designed for clients and CMIs deployed in a data center. Rejig does not require the coordinator to propogate a new configuration to all CMIs. It employs clients to perform this task on demand, making it appropriate for other deployments. For example, it may be used with trusted mobile clients[2] that have intermittent network connectivity to the caching layer. It also functions with CMIs deployed across two or more geographically distributed data centers [3]. These deployments are possible because Rejig satisfies the following properties. First, CMIs are passive entities that respond to requests. Second, clients do not transmit a configuration to one another directly. They use CMIs and the coordinator intelligently to obtain the latest configuration on demand, i.e., once a Rejig client detects that its request was issued using an old configuration.



(a) Hash partitioning.



(b) Range partitioning.

Fig. 2. A configuration with two partitioning techniques.

[2] Untrusted mobile clients may open possibilities for a Denial of Service (DoS) attack or data corruption.

The software architecture of Rejig hides its implementation details. Its transparency frees software developers to focus on their application and its requirements (instead of configuration changes and how to manage impacted fragments). Central to its design is a monotonically increasing global unique identifier for each configuration published by the coordinator, GlobalCfgID. Each updated and inserted cache entry is tagged with the current GlobalCfgID. Moreover, each client must piggyback the GlobalCfgID of its cached configuration with the request it issues to a CMI. Hence, Rejig imposes two types of overhead. First, it requires 4 extra bytes of memory per cache entry to store GlobalCfgID. Second, a client request consumes 4 extra bytes of the available network bandwidth per request to transmit GlobalCfgID of its configuration.

The rest of this paper is organized as follows. Section 2 details Rejig's design. Section 3 presents an implementation of this design. We evaluate this implementation using microbenchmarks and traces from Azure [10] and WorldCup '98 [4] in Sect. 4. Section 5 contains related work. Brief conclusions are presented in Sect. 6.

2 General Design of Rejig

Rejig's software architecture consists of cache manager software, a coordinator responsible for maintaining the configuration, and a client component used by the application to issue requests. Figure 1 shows an example deployment of this software architecture. A cache server hosts one or more Rejig cache manager instances, CMIs. Each CMI is identified using the combination of its IP and port number, see Fig. 2. The coordinator represents a configuration as F fragments assigned to different CMIs. We detail a space efficient implementation of a configuration in Appendix B.

Rejig supports hash and range partitioning techniques to shard the application's data across CMIs. With the former, a hash function is used to identify the fragment containing a referenced key K. With the latter, the configuration maintains the range of values assigned to each fragment. It performs a binary search to identify the fragment containing K. Rejig also supports a hybrid partitioning technique [1] that applies a hash function to map K to an order preserving space that is range partitioned across CMIs. We describe how Rejig's coordinator increases the number of fragments F with the hash partitioning strategy in Appendix B.1.

Rejig's coordinator uses a monotonically increasing integer, GlobalCfgID, to identify a configuration. Rejig's client and server components maintain their latest value of GlobalCfgID. Rejig's client component caches a local copy of the latest configuration for efficient processing of requests. When the coordinator computes a new configuration, it increments GlobalCfgID. For each impacted fragment, the coordinator informs either a subset or all impacted CMIs of the new GlobalCfgID and inserts the corresponding configuration in these CMIs. As an example of updating a subset of the impacted CMIs, consider a scenario that migrates a fragment from a source CMI to a destination CMI. It is sufficient

Table 1. Terms and their Definitions.

Term	Definition
F	Number of fragments
CMI_i	Cache manager instance i
Configuration	Mapping of F fragments to CMIs
F_k	Fragment k
GlobalCfgID	Coordinator maintained monotonically increasing value that identifies a configuration
FragCfgID_k	GlobalCfgID value of the configuration that either created or changed the assignment of fragment F_k
V_{cid}	Configuration id associated with a cache entry when it is either inserted or updated in a CMI
α	Number of CMIs a client inserts GlobalCfgID and configuration into once it obtains the latest configuration

for Rejig to update the source CMI with the new GlobalCfgID and insert the latest configuration in this CMI only. As an example of updating all impacted CMIs, when a CMI is removed, its fragments are assigned to different CMIs. The coordinator updates the GlobalCfgID of these CMIs and inserts the new configuration in all (Table 1).

Rejig's clients and CMIs use a distributed collaborative algorithm to update their GlobalCfgID value and cached configuration. Section 2.1 details this algorithm.

Rejig's coordinator implements a re-organization algorithm that changes the assignment of fragments to CMIs in response to an evolving workload. The efficiency of a configuration change and how it re-organizes fragments' assignment is the responsibility of this algorithm (and not Rejig). At any given time, there is one active coordinator. However, there may be multiple standby coordinators, each of which is prepared to take over if the active coordinator crashes. The active coordinator stores the latest configuration and its GlobalCfgID on an external storage system that is highly available (such as ZooKeeper [26]). The standby coordinators use the external storage system to detect failure of the active coordinator, select a new active coordinator, and recover the configuration and its GlobalCfgID.

With each configuration change, the coordinator maintains the value of GlobalCfgID for each fragment F_k impacted by that change, FragCfgID_k. A fragment's FragCfgID_k is initialized to the GlobalCfgID value that created it. A fragment is impacted by one configuration change. However, a configuration change may impact several different fragments. For example, removal of a cache server in Fig. 2 impacts multiple CMIs and their assigned fragments. Thus, at any instance in time, different fragments may have different FragCfgID_k values, identifying the GlobalCfgID value that changed their assignment to a CMI.

Table 2. Rejig and its variants.

Rejig	A client always inserts its newly obtained configuration in the next α unique CMIs
Rejig^C	A client inserts its newly obtained configuration in the next α unique CMIs only if it fetches the latest configuration from the coordinator
Rejig^T	Similar to Rejig with the following termination condition. A client stops inserting its obtained configuration in CMIs once a CMI reports it has the latest configuration

Example 2.1. In Fig. 2(b), F_j's FragCfgID is 765. Assume the current GlobalCfgID value is also 765, GlobalCfgID = 765. If the coordinator assigns a different fragment, say F_0, to a different CMI then it increments GlobalCfgID by one, GlobalCfgID = 766. It produces a new configuration with F_0's FragCfgID set to 766 along with IP and port number of its newly assigned CMI. Other fragments' FragCfgID including F_j's FragCfgID remain unchanged.■

2.1 Processing Get Requests

Algorithms 1 and 2 provide Rejig's protocol to process the get command by a client and a CMI, respectively. Appendix A provides a formal proof of this protocol. A client piggybacks its value of GlobalCfgID with every request it issues to a CMI_i, see line 3 of Algorithm 1. A cache manager instance, CMI_i, compares its latest known GlobalCfgID to the one provided by the client. There are three possibilities, see lines 1 to 8 of Algorithm 2. Either the two are equal, CMI_i's GlobalCfgID is greater than the client's GlobalCfgID, or the CMI's GlobalCfgID is less than client's GlobalCfgID. Consider each in turn.

CMI_i's GlobalCfgID Equals Client Provided GlobalCfgID: If CMI_i has the value associated with the referenced key then it returns this value including its configuration id, V_{cid}. V_{cid} identifies the configuration in which this cache entry was either inserted or updated in CMI_i (line 10 in Algorithm 2). The client compares V_{cid} with its assigned fragment's $FragCfgID_k$. If V_{cid} is greater than or equal then the value is valid and the client provides it to the application. Otherwise, the value was created in an older configuration that mapped this fragment to the same CMI and the value *may be* stale. Hence, the client discards the value and reports a cache miss. One may configure Rejig clients to delete this cache entry, freeing the available cache space of the CMI, see lines 19 to 24 in Algorithm 1.

This algorithm may incorrectly identify a value as stale if it was not updated while its fragment F_k was assigned to some other CMIs. The likelihood of this false negative is a function of the popularity of the cache entry, the mix of reads and writes in the workload, and how long F_k was mapped to a different CMI before being re-assigned. We quantify this in Sect. 4.2.

Algorithm 1: Client: get

```
get(key)
Input: key: byte array
Result: cached value or null
Let ConfigKey = the key that identifies the cache entry of a configuration.
Let LatestConfig = Client's latest copy of the configuration.
Let ClientGCfgID = Client's current GlobalCfgID value.
1  fragment = getFragment(LatestConfig, key);
2  cache = fragment.CMI;                                 // get the assigned CMI
3  result = cache.get(ClientGCfgID, key);
4  if result.code == RefreshAndRetry then
5      newConfig = cache.get(ConfigKey);
6      if newConfig.value $\neq$ null then
7          LatestConfig = newConfig;
8          ClientGCfgID = newConfig.GlobalCfgID;
9      else
           /* the configuration cache entry may be evicted                  */
10         newConfig = coordinator.getLatestConfig();
11         LatestConfig = newConfig;
12         ClientGCfgID = newConfig.GlobalCfgID;
13     end
14     return get(key);
15 else
16     if cache is one of the next $\alpha$ unique CMIs then
17         cache.set(ClientGCfgID, ConfigKey, LatestConfig);
18     end
19     if result.code == hit then
20         if fragment.FragCfgID$_k$ $\leq$ result.V$_{\text{cid}}$ then
21             return result.value;
22         else
23             cache.delete(ClientGCfgID, key) ;             // asynchronously
24             return null;
25         end
26     else
27         if result.code == miss then
28             return null;
29         end
30     end
31 end
```

CMI$_i$'s GlobalCfgID is Greater than the Client's GlobalCfgID: This condition is satisfied when the client's cached configuration is old and CMI$_i$ is provided with a more recent configuration. CMI$_i$ returns a "*Refresh & Retry*" response to the client, see line 2 in Algorithm 2. In response, the client fetches the latest configuration from CMI$_i$ and retries its request. If CMI$_i$ evicted[3] the latest

[3] CMI$_i$ may pin the latest configuration to prevent its eviction.

configuration (i.e., reports a cache miss), the client contacts the coordinator for the latest configuration. Subsequently, it retries its request, see lines 4 to 14 in Algorithm 1. It is possible to reduce the number of roundtrips by requiring a CMI$_i$ to piggyback the new configuration (assuming CMI$_i$ has it) with its "*Refresh & Retry*" response.

Rejig clients disseminate the latest configuration to one another using the CMIs. A client that fetches a configuration inserts it into the next α unique CMIs that it contacts to process a request, piggybacking its GlobalCfgID value along with each insert, see line 17 in Algorithm 1. A CMI ignores this insertion when its GlobalCfgID is greater, i.e., the configuration changed and this CMI has a more recent configuration.

There are other variations of this dissemination technique [12]. We consider two variants named Rejig^C and Rejig^T, see Table 2. With Rejig^C, a client inserts its known configuration into the next α unique CMIs only if it fetches the latest configuration from the coordinator. With Rejig^T, a client stops inserting once a CMI reports that it has the latest configuration. We quantify the tradeoffs associated with Rejig, Rejig^C, and Rejig^T in Sect. 4.1.

CMI$_i$'s GlobalCfgID is Less than the Client's GlobalCfgID: CMI$_i$ deletes its known configuration and sets its GlobalCfgID with the one provided by the client, see lines 5 to 6 in Algorithm 2. The client may insert its known configuration in CMI$_i$, see line 17 of Algorithm 1.

Algorithm 2: CMI: get

```
get(ClientGCfgID, key)
Input: ClientGCfgID: integer, key: byte array
Result: response code, the cached value and the cache entry's configuration id if
        found.
// ClientGCfgID is the client's GlobalCfgID value
Let ConfigKey = the key of the latest known configuration.
Let CMIGCfgID = CMI's current GlobalCfgID value.
1  if CMIGCfgID > ClientGCfgID then
2      return RefreshAndRetry;
3  else
4      if CMIGCfgID < ClientGCfgID then
5          CMIGCfgID = ClientGCfgID;
6          delete(ConfigKey);
7      end
8  end
9  if key is cached then
10     return cache hit, the cached value and its associated configuration id;
11 end
12 return cache miss;
```

When multiple threads of a client receive "*Refresh & Retry*", only one thread obtains the latest configuration from CMI_i while other threads wait. Once this thread obtains the configuration, all threads use it to retry their requests. Threads referencing CMIs other than CMI_i are not blocked.

In sum, Rejig employs clients (instead of the coordinator) to propagate a new configuration to other CMIs on demand. This prevents the coordinator from becoming a bottleneck, realizing a scalable Rejig protocol.

Example 2.2. Consider the range partitioned configuration of Fig. 2(b). Cache entry K_i is assigned to Fragment F_j. F_j is assigned to CMI 1.0. Assuming GlobalCfgID = 765, a write of K_i sets its $\mathrm{V}_{\mathrm{cid}}$ to 765. Assume a configuration change that assigns F_j to the CMI hosting F_0, CMI 0.5. This results in the following changes: GlobalCfgID is set to 766, F_j's FragCfgID is set to 766, F_j's CMI IP and port number are set to those of CMI 0.5. Another write of K_i is directed to CMI 0.5, creating this cache entry and setting its $\mathrm{V}_{\mathrm{cid}}$ to 766. Now, the copy of K_i on CMI 1.0 is stale. Should another configuration change assign F_j back to CMI 1.0, the GlobalCfgID is incremented by one, GlobalCfgID = 767. Moreover, F_j's FragCfgID is also set to 767 and its IP and port number is set to CMI 1.0. A reference for K_i may observe the stale version. This version's FragCfgID (765) is lower than F_j's FragCfgID 767. Hence, Rejig discards this version and reports a cache miss. ■

2.2 Read-After-Write Consistency

Intuitively, Rejig provides read-after-write consistency for several reasons. First, a CMI impacted by a configuration change does not process a request by a client that does not have a GlobalCfgID pertaining to either this configuration change or a more recent one. Second, a client must update its configuration when a CMI provides a "Refresh & Retry" response to the client request, increasing the response time of this request. This increase is dictated by the amount of time required for the CMI to transmit the new configuration to the client. Third, a CMI always updates its GlobalCfgID to the latest GlobalCfgID provided by a client or the coordinator. Fourth, a cache entry that observes a hit is valid only if its configuration id[4] $\mathrm{V}_{\mathrm{cid}}$ is more recent than the configuration that changed its fragment's assignment. The latter ensures replicated cache entries from a previous configuration (that have not yet been evicted and are potentially stale) are discarded. See Appendix A for a formal proof.

2.3 CMI Discarding Stale Cache Entries

Thus far, a Rejig client discards cache entries identified as potentially stale. A CMI may report a miss instead of transmitting a potentially stale cache entry to a client. To realize this, Rejig's client component is extended to provide both $\mathrm{FragCfgID}_k$ and GlobalCfgID with every read. After a CMI verifies that a client

[4] The configuration that inserted or updated this entry.

provided GlobalCfgID is the latest, it must verify the V_{cid} of the referenced cache entry is greater than or equal to the FragCfgID_k. If this is the case then it provides the cache entry to the client. Otherwise, it reports a cache miss and deletes this entry. This requires extra processing by a CMI and incurs the overhead of transmitting FragCfgID_k with every read request. However, if configuration changes are the norm and discarded cache entries are large in size then this approach may save network bandwidth.

2.4 Leases

Rejig assumes an architecture that uses leases to ensure consistency of data in the presence of server failures and network partitions. These leases are managed by the coordinator of Fig. 1. A lease is similar to a lock but with a fixed lifetime [25]. A CMI may process requests referencing Fragment F_k as long as the coordinator grants it a valid lease on F_k. The CMI may contact the coordinator to renew its lease on F_k prior to its expiration. Once a lease on F_k expires, the CMI stops servicing requests referencing F_k. The coordinator then assigns F_k to other available CMIs.

Similarly, before the coordinator changes the assignment of F_k from CMI_i to CMI_j, it (1) revokes F_k's lease from CMI_i to stop it from processing requests, and (2) grants a lease on F_k to CMI_j to enable it to process requests referencing F_k. Subsequently, it changes the configuration and uses Rejig to propagate the new configuration to the clients.

Leases and Rejig serve different purposes and complement one another. Both are required to implement read-after-write consistency in the presence of network partitions and configuration changes. While leases ensure data availability in the presence of network partitions, Rejig disseminates a new configuration efficiently. In particular, Rejig enables a CMI to use its eviction policy to delete cache entries of those fragments that are no longer assigned to it (lazily as the space occupied by these entries is required).

The coordinator does not publish a new configuration that impacts the assignment of a fragment from/to a CMI that is unreachable. It waits for the lease to expire (or the network connection to be restored to issue a revoke/grant lease) prior to publishing a new configuration.

When a client references a CMI for a cache entry assigned to a fragment with an expired lease, the CMI may respond with a "*Refresh & Retry*" response. This causes the client to look up the CMI for the latest configuration. If the CMI reports a miss, the client contacts the coordinator for the latest configuration. If no CMI is assigned to this fragment then the coordinator selects the least loaded CMI, assigns the lease on the fragment to it, increments its GlobalCfgID and computes a new configuration, updates the GlobalCfgID of the CMI, inserts the latest configuration in the CMI, and provides the client with the latest configuration.

2.5 Replication for High Availability

A system may construct R replicas of a fragment to enhance data availability in the presence of CMI failure(s). Below, we describe two popular approaches to maintain these replicas consistent. For each, we describe how a failure is represented as a configuration change and supported by Rejig.

The first approach requires (1) a read action to obtain r Shared (S) leases prior to reading the value of a replica and (2) a write action to obtain ω eXclusive (X) leases prior to writing all replicas [11]. While S leases are compatible, X leases conflict with one another and S leases. Conflicts cause the leases to race with the loser backing off and retrying. For read-after-write consistency, $r + \omega$ must be greater than R. With this approach, failure of a CMI is a configuration change that removes one replica of a fragment, decrementing R by one.

The second approach designates one copy of a fragment as primary and the other $R - 1$ as secondaries [22,39]. All reads and writes are processed by the primary fragment. The primary is responsible for propagating updates to the secondaries in the same order it receives them. If the primary fails, the coordinator promotes one of its secondaries to become the primary. With this design, a configuration change reflects promotion of a secondary to the primary and demotion of the primary to be a secondary. The coordinator increments GlobalCfgID. The value of FragCfgID$_k$ for the promoted secondary is left unchanged. If this new primary buffers changes for the demoted primary, then the value of FragCfgID$_k$ for the demoted primary is also left unchanged. Should the coordinator decide to discard the fragment of the demoted primary, then it simply sets its FragCfgID$_k$ to the latest value of GlobalCfgID.

Coordinator inserts the new configuration into $R-1$ secondaries and updates their GlobalCfgID. A client that fails to issue a request to the failed primary CMI may contact one of the secondary CMIs for the latest configuration. If it observes a miss then the client contacts the coordinator for the latest configuration piggybacking the identity of the unavailable CMI. Hence, clients discover failed CMIs and report them to the coordinator.

2.6 Overhead

Rejig's overhead is in the form of storage space and network bandwidth. Storage overhead include (1) a 4-byte configuration id associated with a cache entry, (2) a configuration cache entry in a CMI. Network overhead include transmission of (1) a 4-byte client configuration id attached to each request, (2) at most two round-trips per client to get the latest configuration (clients first retrieve the configuration from a CMI and, if not found, fetch the configuration from the coordinator), (3) α roundtrips per client to insert a new configuration into CMIs.

We quantify Rejig's overhead based on the statistical models of Facebook's production key size and value size [5,19]. Facebook's mean key size is 35 bytes and the mean value size is 329 bytes, resulting in the average entry size of 364 bytes. Hence, the average storage overhead of the configuration id associated with a cache entry is 1% (4/364). With a configuration consisting of F = 5000 fragments and 12-byte metadata of each fragment, the configuration size is 60,000 bytes. This is equivalent to 165 (60,000/364) of Facebook's cache entries. Rejig's network overhead is 11.5% (4/35) for get and delete requests, and 1% (4/364) for SET requests with Facebook's cache entries.

3 Implementation

We implemented a prototype of Rejig using memcached's Whalin client [41]. Its client library is implemented with around 500 lines of Java code. Client interfaces to communicate with a CMI is unchanged.

We implemented Rejig's coordinator using Google RPC [24] with 800 lines of Java code. The coordinator increases or decreases the number of CMIs based on the system load and adjusts the assignment from fragments to CMIs with the goal to ensure each CMI receives a similar number of fragments.

We extended IQ-Twemcached [21] (the Twitter extended version of memcached with Inhibit and Quarantine leases) to store GlobalCfgID and associate each cache entry with a configuration id. We extended standard APIs, e.g., get, set, and delete, to accept an optional configuration id. We also added a new API in IQ-Twemcached to allow the coordinator to update a CMI's configuration id. The extension to the IQ-Twemcached only requires 40 lines of C code.

4 Evaluation

We answer the following questions in this section: *(1) How fast can Rejig disseminate a new configuration? (2) How much stale data does Rejig prevent? (3) Does Rejig impact cache hit rate in the presence of graceful and drastic configuration changes?*

Factors that impact Rejig's dissemination rate are the number of clients and CMIs. CMIs are passive providers of a new configuration. Once all CMIs receive a new configuration, all clients receive the new configuration as soon as they issue a request to a CMI. The load imposed on the coordinator and the number of configuration insertions to CMIs are an interplay of α, Rejig and its variants, and the duration of time a CMI caches a configuration. In the worst case scenario, the size of a configuration cache entry is larger than the available memory of each CMI, causing the load imposed on the coordinator to equal the number of clients. In the best case scenario, each CMI has sufficient memory to cache the configuration and never evicts it. In this case, a larger α expedites dissemination of the configuration to all CMIs, reducing the number of clients that fetch the configuration from the coordinator. Rejig^C bounds the number of configuration fetches from the coordinator to be the number of CMIs. This is because a client must discover a new configuration from a CMI and fetches the configuration from the coordinator before the client propagates the new configuration to other CMIs. Rejig^T bounds the number of repeated configuration insertions in CMIs to the number of clients since a client terminates the insertion once it encounters a CMI with a copy of the configuration. Section 4.1 quantifies Rejig's dissemination rate.

We use Azure virtual machine event traces [10] and WorldCup '98 request traces [4] to demonstrate the number of stale data Rejig prevents and its impact on the cache hit rate. Azure trace exhibits graceful configuration changes while WorldCup '98 trace exhibits drastic configuration changes with a diurnal pattern.

Lastly, we use YCSB [8] benchmark to evaluate the performance impact of Rejig, namely, tagging a request with the client's configuration id and returning a cache entry's configuration id along with its value upon a cache hit. Our evaluation shows that Rejig's network overhead is insignificant. The average read latency increases by less than 2% with Rejig when compared to without it.

Rejig can also be extended to support diverse migration techniques [27,33,36,42]. In its current format, once the coordinator assigns a fragment to a different CMI, the new CMI starts with an empty replica of the fragment. If migration is enabled, a cache miss on the new CMI migrates the cache entry from the original CMI.

Main lessons are as follows:

- Rejig disseminates a new configuration to all clients and CMIs efficiently and quickly.
- Collaborative dissemination reduces the number of clients that contact the coordinator for the latest configuration significantly.
- Rejig preserves read-after-write consistency in the presence of configuration changes by detecting and discarding all consistency violations.
- In all experiments, the cache hit rate remains high even though a configuration change does not migrate cache entries. With the trace driven analysis, the cache hit rate is always higher than 99%.
- With $\alpha \geq 1$, if the objective is to minimize the number of configuration fetches from the coordinator then Rejig is superior to the alternatives shown in Table 2. On the other hand, if the objective is to minimize the number of repeated configuration insertions into CMIs then Rejig^C is a superior technique. Rejig^C is superior to Rejig^T.

Below, we describe experiments in turn.

4.1 Scalable Configuration Dissemination

We design a microbenchmark to evaluate Rejig's configuration dissemination. It consists of 100 CMIs, a fixed number of clients, and one coordinator. There are 5000 fragments, F = 5000. We use Facebook's published cache entry size of 364 bytes [5]. The size of a configuration is 60,000 bytes, F × 12; twelve bytes assuming 8 bytes for CMI address + 4 bytes for FragCfgID. For experiments of Sects. 4.1 and 4.1, we assume the total available memory is greater than the database size and there are no evictions. We consider limited memory in Sect. 4.1.

An experiment performs a sequence of iterations. In each iteration, each client issues a read for a randomly selected key K_i assigned to CMI_i,

i = Config[h(K_i)].CMI. The experiment starts with the coordinator publishing a new configuration that assigns a fragment from a source CMI to a destination CMI. The coordinator inserts the new configuration into the source CMI. The experiment terminates when all clients and CMIs have the latest GlobalCfgID and configuration. We report the number of iterations required for Rejig to disseminate a configuration change.

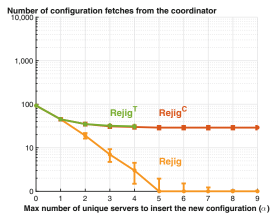
Configuration Dissemination Rate. The number of clients impacts how fast Rejig disseminates a configuration change. As we increase the number of clients from 100 to 1,000 and 10,000, Rejig requires 12 ± 1.7, 5 ± 0.5, 4 ± 0 (mean $\pm$ standard deviation) iterations, respectively. This is also an upper bound on the number of client configuration insertions into a CMI. For a CMI to receive the latest GlobalCfgID, a client with the latest GlobalCfgID must issue a request to it. With a larger number of clients, a larger number of requests are issued to CMIs in each iteration. This increases the likelihood of clients referencing all CMIs in an iteration, causing the experiment to terminate with fewer iterations. The reported number of iterations is orthogonal to the value of α because a CMI always updates its configuration id if the client provided one is more recent.

The number of configuration fetches from the coordinator depends on the value of α. With $\alpha = 0$, it is approximately the same as the number of clients: 93 ± 2.2, 962 ± 5.4, 9745 ± 16.3 with 100, 1000, and 10,000 clients, respectively. The explanation for this is as follows. Once the coordinator publishes the new configuration in one CMI, say CMI$_i$, it also notifies this CMI of the new GlobalCfgID. A client that references CMI$_i$ and obtains the latest GlobalCfgID will subsequently reference another CMI$_j$ and cause its GlobalCfgID to reflect the latest *without* providing it with the latest configuration. Those clients that reference this CMI$_j$ observe a cache miss for the configuration and fetch the configuration from the coordinator. The number of configuration fetches is slightly less than the number of clients because those that observe a hit for the configuration using CMI$_i$ do not contact the coordinator.

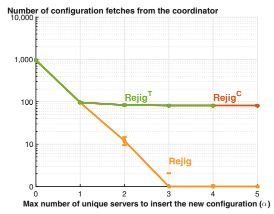
Next section discusses $\alpha \geq 1$.

Collaborative Dissemination, $\boldsymbol{\alpha \geq 1}$. In a second experiment, we evaluate the impact of requiring a client to insert a new configuration in its next unique $\alpha \geq 1$ referenced CMIs. This is the standard Rejig. We consider its variants Rejig^C and Rejig^T, see Table 2. Results of this section show Rejig^C is superior to Rejig^T.

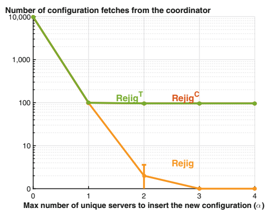
Figure 3 shows the number of configuration fetches with these variants as a function of α. With Rejig, the number of configuration fetches from the coordinator drops 100 folds with $\alpha = 1$ and 10,000 clients when compared with $\alpha = 0$, see Fig. 3(c). When a client inserts its configuration in the next $\alpha = 1$ unique CMI it visits, other clients that reference CMI$_j$ and observe a "Refresh & Retry" reply will now observe a cache hit for the configuration. Hence, they will *not* fetch the configuration from the coordinator. The number of configuration fetches from the coordinator continues to drop as we increase the value of α.



(a) 100 clients.

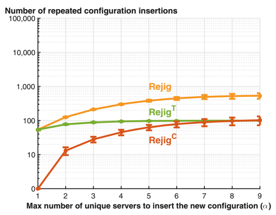


(b) 1,000 clients.

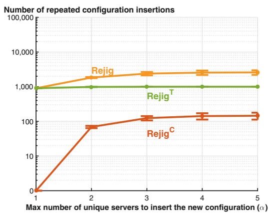


(c) 10,000 clients.

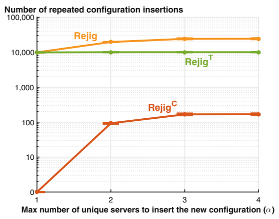
Fig. 3. The impact of α on the number of configuration fetches from the coordinator.



(a) 100 clients.



(b) 1,000 clients.



(c) 10,000 clients.

Fig. 4. The impact of α on the number of repeated configuration insertions in CMIs.

With Rejig and $\alpha \geq 4$, there are almost no client fetches from the coordinator. Clients and CMIs facilitate propagation of the configuration among themselves without further involvement of the coordinator.

With Rejig^C, the number of configuration fetches from the coordinator is higher than 10 as we increase α beyond 4. Rejig^C requires a client to insert a configuration only if it fetches the configuration from the coordinator. Hence, it is less aggressive in spreading the latest configuration when compared with Rejig. This increases the likelihood of a client observing a miss for a configuration in a CMI. Hence, the number of configuration fetches from the coordinator is higher.

A larger α causes two or more clients to insert the same configuration into the same CMI repeatedly. Figure 4 shows the total number of repeated insertions with different variants. Rejig^C performs the fewest repeated insertions because it is the least aggressive. The standard Rejig performs the most because every client that obtains the latest configuration (either from the coordinator or a CMI) will insert into the next α unique CMIs. Rejig^T is moderately aggressive by requiring a client to terminate configuration insertion once a CMI reports that it has the configuration.

These results show Rejig^C is superior to Rejig^T for two reasons. First, it performs significantly fewer configuration insertions than Rejig^T, see Fig. 4. Second, its overall number of configuration fetches from the coordinator is comparable, see Fig. 3.

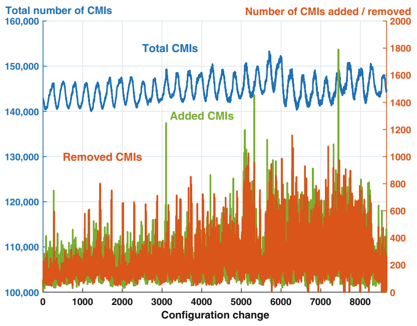
Worst Case Scenario. Consider a worst case scenario where the size of a configuration equals the amount of memory assigned to each CMI. (This is highly unlikely because the size of a configuration is in the order of hundreds of kilobytes with thousands of nodes and memory sizes are typically much larger.) Thus, the coordinator's insertion of a configuration evicts all cached entries of that CMI. Similarly, a client that inserts a cache entry upon a cache miss will evict the configuration cache entry. With $\alpha = 0$, almost all clients (9998) fetch the configuration from the coordinator. The coordinator populates at least one CMI and this copy is fetched by 1 or 2 clients. As we increase α, the number of configuration fetches drops dramatically; 9998 ± 0.15, 2197 ± 133.93, 110 ± 10.39 and 24 ± 5.26 for α values of 0, 1, 2 and 3, respectively. This is because a client's insertion of the configuration in a CMI may observe a reference by another client prior to an entry's insertion that evicts the configuration. The likelihood of this hit increases with a larger values of α, reducing the number of configuration fetches from the coordinator.

4.2 Trace Driven Evaluation

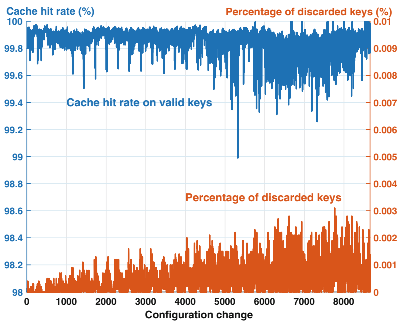
We use two traces to evaluate Rejig: Azure virtual machines (VM) trace [10] and 92 days of WorldCup 1998 request trace [4]. The first trace provides a dynamic addition and removal of VMs. We augment it with a database and use its trace to emulate addition and removal of CMIs from a configuration. WorldCup 1998 provides request traces (HTTP GET/POST on a page) with approximately 1.3 billion requests. It exhibits drastic workload fluctuations. We augment it with an auto-scaling framework that adjusts addition (Add_i) and removal (Remove_i) of CMIs based on the imposed load.

With both traces, when Add_i new CMIs are inserted in a configuration, the coordinator assigns fragments of existing CMIs to the new CMIs until the number of fragments per CMI is approximately the same. No data is migrated. Similarly, when Remove_i CMIs are removed, the coordinator assigns the orphaned fragments to other CMIs until the same number of fragments are assigned to each CMI. Once again, no data is migrated.

In all experiments, the values stored in a CMI are known and the workload generator can verify the correctness of the fetched values. We establish the following metrics: (1) the cache hit rate, (2) the number of discarded keys, and (3) the percentage of discarded keys. The percentage of discarded keys highlights the percentage of read requests that may observe stale entries. If a client produces a discarded key as an output, then the read request may violate read-after-write consistency. The number of invalid discarded entries highlights how many read requests violate read-after-write consistency with a system that does not use our techniques. Rejig eliminates these stale entries. We describe each trace and obtained results in turn.



(a) Added and removed CMIs.



(b) Hit rate and percentage of discarded keys.

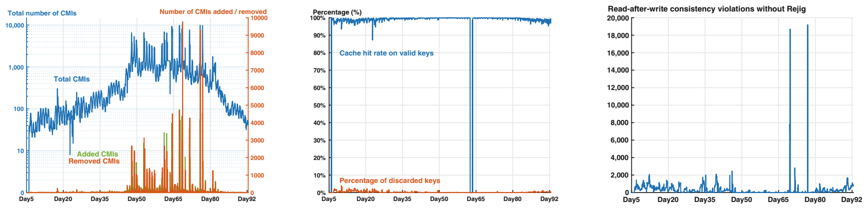
Fig. 5. Azure trace.

Azure VM Trace. Azure virtual machines (VM) trace [10] provides a representative subset of the first-party Azure VM workload in one geographical region. It monitors VMs in a consecutive 30-day period and contains a total of 2 million VMs. It provides the exact lifetime of each virtual machine with approximately 145,000 VMs running at a time. This trace has no data set. Hence, we generate a database with 10 million key-value records partitioned into 100,000 fragments for our evaluation using this trace. Before a configuration change, we randomly select 20% of entries and update their values. Since a configuration change does not migrate data, copies of these entries in the impacted CMIs are now stale.

Figure 5(a) shows the number of configuration changes we extracted from the Azure trace. It shows both the number of added CMIs, removed CMIs, and total CMIs per configuration change. Figure 5(b) shows both the observed cache hit rate and the percentage of keys discarded by Rejig with each configuration. The cache hit rate remains higher than 99% even though a configuration change may add 1800 CMIs. The variation in cache hit rate is higher with a larger number of CMIs either added or removed from a configuration. This is expected because a higher number of fragments are assigned to different CMIs that results in more cache misses. Note the percentage of discarded keys is low (less than 0.003%) in this experiment. Among all discarded keys, 99% are stale. These are read-after-write inconsistencies prevented by Rejig.

WorldCup 1998 Trace. WorldCup 1998 [4] exhibits a diurnal pattern and drastic workload fluctuations. It contains 92 days of request traces at the granularity of seconds. A large percentage (99.98%) of its approximately 1.3 billion requests are reads. Peak system load observes 10 million requests per hour. The traces reference 89,997 unique keys. We start the simulation from Day 5 (The first four days have no data).

We use the following auto-scaling framework with this trace. We assume the peak processing capacity of a CMI is $C = 1000$ requests per second. We define



(a) Added, removed, and total CMIs. (b) Hit rate and percentage of discarded keys. (c) Rejig avoided read-after-write violations.

Fig. 6. WorldCup 1998 trace.

the imposed system load as the number of requests at a given time, Load_i. The number of required CMIs is a function of these two parameters, the number of CMIs to be added $\text{Add}_i = \max(\frac{\text{Load}_i}{C} - \text{S}_{i-1},\ 0)$ and the number of CMIs to be removed $\text{Remove}_i = \max(\text{S}_{i-1} - \frac{\text{Load}_i}{C},\ 0)$. These ensure the caching layer has sufficient number of CMIs to handle Load_i at a given hour i. This auto-scaling is performed every hour of the trace for its entire 92 days.

Figure 6(a) reports the total number of added and removed CMIs per hour. As expected, more CMIs are added during the daytime. They are removed during the nighttime. The resulting drastic configuration change causes the cache hit rate to fluctuate as we do not migrate data, see Fig. 6(b). However, the percentage of discarded keys remains low. The 0% cache hit rate on Day 60 is because there is no data for a few hours on that day. The number of read-after-write consistency anomalies avoided by Rejig is still significant even though the update ratio is only 0.02%, see Fig. 6(c).

5 Related Work

Existing work [20,21,33] on cache augmented database systems focus on eliminating or minimizing inconsistency between the caching layer and the data store. IQ-framework [21] uses leases to ensure a cache entry's replica in the caching layer is consistent with its replicas in the data store. Gumball [20] minimizes inconsistency by rejecting reads and writes referencing a cache entry for a certain duration after the entry is deleted. Facebook [33] sets this duration to two seconds. Rejig complements these systems by preserving consistency in the presence of configuration changes.

The CAP theorem [23] states that a distributed data store must choose two out of the three properties: consistency, availability, and partition tolerance. Rejig does not address these properties directly. Rejig is a scalable online algorithm for cache server configuration changes. Rejig assumes an architecture that uses heartbeat messages to detect network partitions and leases to re-claim fragments assigned to unreachable nodes. While it strives to preserve consistency, one may use Rejig with an architecture that uses eventual consistency.

Two-phase commit [37] based systems (e.g., Sinfonia [2]) and Paxos [29] based systems (e.g., Spanner [9]) ensure consistency between multiple replicas. Rejig does not ensure consistency across all replicas. Instead, it detects and discards stale cache entries caused by configuration changes.

Rejig's configuration dissemination protocol is inspired by Demers et al.'s work [12] on epidemic algorithms where all sites of a data store actively participate in propagating an update made by one site. Rejig propagates a configuration made by the coordinator using both clients and CMIs. Its CMIs are passive entities that respond to client requests. Its clients actively propagate configuration to other clients using CMIs. Rejig's dissemination protocol is also applicable to a data store.

Rejig's configuration management is inspired by previous work, e.g., Google File System [22], Hyperdex [13], and Slicer [1]. Rejig is unique because it is designed for a caching solution and not a data store. With Rejig, there is a permanent copy of data elsewhere and loss of cached data does not result in data loss. Another novel feature of Rejig is that it is intended for frameworks that allow for stale cache entries to exist. Rejig detects these entries by storing a configuration id with each cache entry and its fragment, see Sect. 2. This concept is missing from prior work. Below, we provide an overview of each related system and how Rejig is different.

Google File System (GFS) [22] is a distributed file system. A GFS file contains a list of chunks. Each chunk is associated with a chunk version number. The master maintains the latest chunk version number for each chunk. Once a chunk server recovers from a failure, the master detects a stale chunk if its version number is less than the master's chunk version number. Rejig stores configuration id with each cache entry and fragment to detect stale cache entries.

Hyperdex [13] is a novel distributed key-value store that supports index structures on more than one attribute. Similar to Rejig, it employs a coordinator that manages its configuration with a strictly increasing configuration id. Upon a configuration change, the coordinator increments the configuration id and distributes the latest configuration to **all** servers. Both Hyperdex servers and its clients cache the configuration id. A client embeds its local configuration id on every request to a server and discovers its cached configuration is stale if the id does not match the server's configuration id. Hyperdex is a data store and prevents stale values for data items. Rejig is for a caching environment where stale entries may exist. It assigns a configuration id to every cache entry and uses this information to detect stale entries and discard them to provide read-after-write consistency.

Slicer [1] is Google's general purpose sharding service that is transparent to its applications. It maintains assignments using generation numbers (equivalent to Rejig's GlobalCfgID). Slicer employs leases to ensure that a key is assigned to one slicelet (equivalent to Rejig's CMI) at a time. Applications are unavailable for at most 4 s during an assignment change due to updating leases to reflect the latest generation number (and assignment) to slicelets. Also, Slicer must provision resources for a large number of distributors to disseminate a new assignment to clients and slicelets. Rejig is designed for caches and is different in several ways. First, while its CMIs may cache a copy of configuration (Slicer's assignment), they do not use it to decide whether to process a client request or not. CMIs use GlobalCfgID (Slicer's generation number) for this purpose. Moreover, Rejig requires its coordinator to update impacted CMIs only and employs both clients and CMIs to participate in distributing a new configuration. Finally, Slicer does not either prevent or detect stale data. Rejig is novel because it detects stale cache entries by storing the configuration id with each entry and fragment.

6 Conclusion

Rejig is a scalable online algorithm for cache server configuration changes that preserves read-after-write consistency. It does not require deletion of cached entries impacted by a configuration change, leaving them to be evicted by the cache replacement technique. It serves as the building block for a fragment reorganization algorithm to balance system load, an auto-scaling framework that grows and shrinks the size of a caching layer, a data availability technique that re-assigns fragments in response to network partitions, and a persistent caching layer [18] that must recover cached entries after a failure.

Acknowledgments. We thank the anonymous reviewers of the TLDKS journal for their valuable comments and helpful suggestions.

Appendix

A Proof for Read-After-Write Consistency

This section presents a formal proof for read-after-write consistency with Rejig.

Theorem 1. *Rejig preserves read-after-write consistency for a cache entry represented as (K, V) mapped to a fragment* F_i *across N configurations* $\mathrm{Config}_i, i \in [1, N]$.

The coordinator creates F_i at Config_1. At a configuration Config_p, $p \in [1, N]$, the fragment F_i is assigned to a cache instance $\mathrm{CMI}_{i,p}$. At configuration Config_1, the coordinator's global configuration id is one and F_i's configuration id is also one. Initially, (K, V) does not exist in $\mathrm{CMI}_{i,1}$.

Lemma 1. *Rejig preserves read-after-write consistency for a cache entry (K, V) if* F_i*'s assigned* CMI *remains the same from* Config_1 *to* $\mathrm{Config}_j, j \in [1, N)$, *i.e.,* $\forall p, p \in [1, j], \mathrm{CMI}_{i,p} = \mathrm{CMI}_{i,1}$.

Proof. Since F_i remains on $\mathrm{CMI}_{i,1}$, its fragment id remains one for all configuration changes from 1 to j. Every entry (K, V) is tagged with the configuration id $\mathrm{V}_{\mathrm{cid}}$ that sets its value. When a write inserts or updates K (belonging to $\mathrm{CMI}_{i,1}$) at a configuration 1 to j, its $\mathrm{V}_{\mathrm{cid}}$ is set to the configuration id of $\mathrm{CMI}_{i,1}$. A read is able to consume (K, V) because its $\mathrm{V}_{\mathrm{cid}}$ is greater than or equal to 1, the configuration id of F_i. ■

Corollary 1. *Rejig preserves read-after-write consistency for a cache entry (K, V) in* $\text{CMI}_{i,j}$ *at* Config_j *if K does not exist in* $\text{CMI}_{i,j}$ *initially.*

Lemma 2. *Rejig preserves read-after-write consistency for a cache entry (K, V) if* F_i*'s assigned CMI changes for the first time at* Config_{j+1}*, i.e.* $\text{CMI}_{i,j} \neq \text{CMI}_{i,j+1}, \forall p, p \in [1, j], \text{CMI}_{i,p} = \text{CMI}_{i,1}$.

Proof. According to Lemma 1, Rejig preserves read-after-write consistency for K from Config_1 to $\text{Config}_j, j \geq 1$. During the configuration change from Config_j to Config_{j+1}, we have

1. the coordinator changes $\text{CMI}_{i,j}$'s configuration id to $j + 1$
2. a client's local configuration id is still $c, c \leq j$.

A client request that references K is directed to $\text{CMI}_{i,j}$. $\text{CMI}_{i,j}$ rejects the request since $j + 1 > c$. Then, the client fetches the latest configuration and issues its request to $\text{CMI}_{i,j+1}$. At configuration Config_{j+1}, Corollary 1 shows that Rejig preserves read-after-write consistency. ■

Lemma 3. *Rejig preserves read-after-write consistency for a cache entry (K, V) at* Config_q *if* $\exists o, p, q, o < p < q \leq N, \text{CMI}_{i,o} = \text{CMI}_{i,q}$ *and* $\text{CMI}_{i,o} \neq \text{CMI}_{i,p}$.

Proof. At configuration Config_q, there are two cases:

Case I: If (K, V) is inserted in $\text{CMI}_{i,o}$ at Config_o, updated at Config_p, and still exists in $\text{CMI}_{i,q}$ at Config_q. Since $\exists o, p, q, o < p < q \leq N, \text{CMI}_{i,o} = \text{CMI}_{i,q}$ and $\text{CMI}_{i,o} \neq \text{CMI}_{i,p}$, F_i's configuration id at $\text{Config}_q >$ F_i's configuration id at $\text{Config}_p >$ F_i's configuration id at Config_o. Then, the configuration id associated with (K, V) in $\text{CMI}_{i,q}$ must be lower than F_i's configuration id at Config_q. A read request that references K at Config_q discards the entry.

Case II: If K does not exist in $\text{CMI}_{i,q}$, Corollary 1 proves Rejig preserves read-after-write consistency. ■

B Physical Representation of a Configuration

A configuration consists of F fragments where F may be significantly larger than the number of CMIs. It is undesirable to repeat the CMI's IP address and port number as it increases the size of the configuration and its serialized representation, and time to serialize and deserialize a configuration. One approach to address this is to maintain the IP address and port number of each CMI in a separate array. Each element of a configuration representing a fragment stores index of the array element corresponding to its assigned CMI. Representing a CMI array element as a short (two bytes) accommodates a maximum of 65,536 CMIs. With 1000 fragments assigned to the same CMI, the memory footprint would be 2000 bytes. This is more compact than repeating IP and port numbers a thousand times. While it makes the software to serialize and deserialize a configuration more complex, it reduces network transmission time of a configuration.

B.1 Configuration Changes that Modify the Number of Fragments

A re-assignment algorithm may break a fragment into q fragments or merge p fragments into one fragment, changing the number of fragments F. These are trivial with range partitioning because they translate into breaking a sub-range into q sub-ranges and merging p adjacent ranges into one, respectively. In its extreme, breaking a fragment may result in sub-ranges that correspond to points. Each point may consist of only one data item. This is justified when the data item is extremely hot [38].

With hash partitioning, when a hash function depends on the value of F, breaking and merging of fragments must be done in a manner that is consistent with the hash function. As an example, assume a simple mod function as the hash function, $\mathrm{h}(\mathrm{K}_i) = \mathrm{K}_i \ \% \ \mathrm{F}$. Incrementing (or decrementing) the value of F by one would re-assign key-value pairs across **all** fragments. Rejig does not support these configuration changes. (Rejig supports re-assignment of fragments to CMIs only.) To use Rejig, one must modify the value of F in a manner that changes the assignment of key-value pairs for the impacted fragment only. This would be similar to extendible [14] and linear [30] hashing algorithms. For example, with the mod hash function, to break a fragment into two, F should double. This would generate a buddy for each existing fragment. The buddy of a fragment is assigned to the same CMI as the fragment. The buddy of the fragment that is broken into two is assigned to a different CMI. To merge two fragments into one, we would change the assignment of its buddy to be the same CMI as the fragment. Subsequently, we scan the array to detect if a fragment and its buddy are assigned to the same CMI. If this is the case then we halve F, $\mathrm{F} = \frac{F}{2}$.

References

1. Adya, A., et al.: Slicer: auto-sharding for datacenter applications. In: 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, pp. 739–753. USENIX Association, Savannah (2016)
2. Aguilera, M.K., Merchant, A., Shah, M., Veitch, A., Karamanolis, C.: Sinfonia: a new paradigm for building scalable distributed systems. In: Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles, SOSP 2007, pp. 159–174. ACM, New York (2007)
3. Annamalai, M., et al.: Sharding the shards: managing datastore locality at scale with Akkio. In: 13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, pp. 445–460. USENIX Association, Carlsbad (2018)
4. Arlitt, M., Jin, T.: A workload characterization study of the 1998 world cup web site. Netw. Mag. Global Internetwkg. **14**(3), 30–37 (2000)
5. Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., Paleczny, M.: Workload analysis of a large-scale key-value store. In: SIGMETRICS, pp. 53–64. ACM, New York (2012)
6. Beckmann, N., Chen, H., Cidon, A.: LHD: improving cache hit rate by maximizing hit density. In: 15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, pp. 389–403. USENIX Association, Renton (2018)

7. Cidon, A., Eisenman, A., Alizadeh, M., Katti, S.: Cliffhanger: scaling performance cliffs in web memory caches. In: 13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, pp. 379–392. USENIX Association, Santa Clara (2016)
8. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, pp. 143–154. ACM, New York (2010)
9. Corbett, J.C., et al.: Spanner: Google's globally-distributed database. In: 10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, pp. 261–264. USENIX Association, Hollywood (2012)
10. Cortez, E., Bonde, A., Muzio, A., Russinovich, M., Fontoura, M., Bianchini, R.: Resource central: understanding and predicting workloads for improved resource management in large cloud platforms. In: Proceedings of the 26th Symposium on Operating Systems Principles, SOSP 2017, pp. 153–167. ACM, New York (2017)
11. DeCandia, G., et al.: Dynamo: Amazon's highly available key-value store. In: SOSP (2007)
12. Demers, A., et al.: Epidemic algorithms for replicated database maintenance. In: Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing, PODC 1987, pp. 1–12. ACM, New York (1987)
13. Escriva, R., Wong, B., Sirer, E.G.: HyperDex: a distributed, searchable key-value store. In: Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, SIGCOMM 2012, pp. 25–36. ACM, New York (2012)
14. Fagin, R., Nievergelt, J., Pippenger, N., Strong, H.R.: Extendible hashing - a fast access method for dynamic files. ACM Trans. Database Syst. **4**(3), 315–344 (1979)
15. T. A. S. Foundation. Apache Ignite (2018). https://ignite.apache.org/
16. Ghandeharizadeh, S., Almaymoni, M., Huang, H.: Rejig: a scalable online algorithm for cache server configuration changes. In: Proceedings of the ACM Symposium on Cloud Computing, SoCC 2018, p. 513. ACM, New York (2018)
17. Ghandeharizadeh, S., et al.: A demonstration of KOSAR: an elastic, scalable, highly available SQL middleware. In: Proceedings of the Posters & Demos Session, Middleware Posters and Demos 2014, pp. 23–24. ACM, New York (2014)
18. Ghandeharizadeh, S., Huang, H.: Gemini: a distributed crash recovery protocol for persistent caches. In: Proceedings of the 19th International Middleware Conference, Middleware 2018, pp. 134–145. ACM, New York (2018)
19. Ghandeharizadeh, S., Huang, H.: Hoagie: a database and workload generator using published specifications. In: Second IEEE International Workshop on Benchmarking, Performance Tuning and Optimization for Big Data Applications, December 2018
20. Ghandeharizadeh, S., Yap, J.: Gumball: a race condition prevention technique for cache augmented SQL database management systems. In: Proceedings of the 2nd ACM SIGMOD Workshop on Databases and Social Networks, DBSocial 2012, pp. 1–6. ACM, New York (2012)
21. Ghandeharizadeh, S., Yap, J., Nguyen, H.: Strong consistency in cache augmented SQL systems. In: Proceedings of the 15th International Middleware Conference, Middleware 2014, pp. 181–192. ACM, New York (2014)
22. Ghemawat, S., Gobioff, H., Leung, S.-T.: The Google file system. In: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, SOSP 2003, pp. 29–43. ACM, New York (2003)
23. Gilbert, S., Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News **33**(2), 51–59 (2002)

24. Google: Google Protocol Buffer (2018). https://developers.google.com/protocol-buffers
25. Gray, C., Cheriton, D.: Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In: Proceedings of the Twelfth ACM Symposium on Operating Systems Principles, SOSP 1989, pp. 202–210. ACM, New York (1989)
26. Hunt, P., Konar, M., Junqueira, F.P., Reed, B.: ZooKeeper: wait-free coordination for internet-scale systems. In: Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference, USENIXATC 2010, p. 11. USENIX Association, Berkeley (2010)
27. Hwang, J., Wood, T.: Adaptive performance-aware distributed memory caching. In Proceedings of the 10th International Conference on Autonomic Computing, ICAC 2013, pp. 33–43. USENIX, San Jose (2013)
28. IBM: IBM WebSphere (2018). https://www.ibm.com/cloud/websphere-application-platform
29. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16**(2), 133–169 (1998)
30. Litwin, W.: Readings in database systems. Chapter Linear Hashing: A New Tool for File and Table Addressing, pp. 570–581. Morgan Kaufmann Publishers Inc., San Francisco (1988)
31. Lu, H., et al.: Existential consistency: measuring and understanding consistency at Facebook. In: Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, pp. 295–310. ACM, New York (2015)
32. Memcached (2018). https://memcached.org/
33. Nishtala, R., et al.: Scaling Memcache at Facebook. Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, pp. 385–398. USENIX, Lombard (2013)
34. Oracle: Oracle Coherence (2018). http://www.oracle.com/technetwork/middleware/coherence/overview/index.html
35. Redis (2019). https://redis.io/
36. Redis migrate command (2019). https://redis.io/commands/migrate
37. Skeen, D., Stonebraker, M.: A formal model of crash recfovery in a distributed system. IEEE Trans. Softw. Eng. **9**(3), 219–228 (1983)
38. Taft, R., et al.: E-store: fine-grained elastic partitioning for distributed transaction processing systems. Proc. VLDB Endow. **8**(3), 245–256 (2014)
39. van Renesse, R., Schneider, F.B.: Chain replication for supporting high throughput and availability. In: OSDI (2004)
40. Waldspurger, C., Saemundsson, T., Ahmad, I., Park, N.: Cache modeling and optimization using miniature simulations. In: 2017 USENIX Annual Technical Conference, USENIX ATC 2017, pp. 487–498. USENIX Association, Santa Clara (2017)
41. Whalin, G., Wang, X., Li, M.: Memcached Whalin Client (2018). https://github.com/gwhalin/Memcached-Java-Client
42. Zhu, T., Gandhi, A., Harchol-Balter, M., Kozuch, M.A.: Saving cash by using less cache. In: 4th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2012. USENIX, Boston (2012)