Gemini: A Distributed Crash Recovery Protocol for Persistent Caches[*]

Shahram Ghandeharizadeh, Haoyu Huang

Database Laboratory Technical Report 2018-06

Computer Science Department, USC

Los Angeles, California 90089-0781

{shahram,haoyuhua}@usc.edu

Abstract

Gemini is a distributed crash recovery protocol for persistent caches. When a cache instance fails, Gemini assigns other cache instances to process its reads and writes. Once the failed instance recovers, Gemini starts to recover its persistent content while using it to process reads and writes immediately. Gemini does so while guaranteeing read-after-write consistency. It also transfers the working set of the application to the recovering instance to maximize its cache hit ratio. Our evaluation shows that Gemini restores hit ratio two orders of magnitude faster than a volatile cache. Working set transfer is particularly effective with workloads that exhibit an evolving access pattern.

1 Introduction

Modern web workloads with a high read to write ratio enhance the performance of the underlying data store with a caching layer, e.g., memcached [25] and Redis [6]. These caches process application reads [26] much faster by looking up results of queries instead of processing them.

A volatile cache loses its data upon a power failure or a process restarts [12]. Upon recovery, the cache provides sub-optimal performance until it rebuilds the application's working set (defined as the collection of data items that the application references repeatedly [9]). The time to restore hit ratio depends on the size of the working set, the time to compute a cache entry and write it to an instance. Yiying et al. [34] report hours to days to warm up cold caches for typical data center workloads.

Existing work on persistent caches [32, 24, 30, 11] recovers cache entries intact from a failure without recovering their latest state, producing stale data. Stale data are undesirable because they defy a user's expectation and increase programming complexity to reason and handle all the complex cases [23]. A crash recovery protocol should preserve read-after-write

[*]A shorter version of this paper appeared in 19th International Middleware Conference (Middleware '18), Rennes, France, December, 2018.

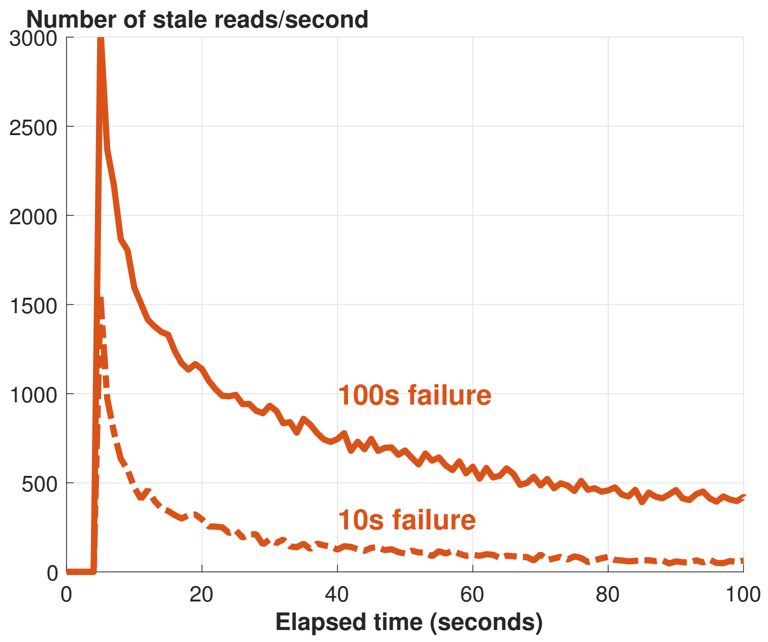


Figure 1: Number of stale reads observed after 20 cache instances recover from a 10-second and a 100-second failure.

consistency [23] even in the event of failure. The caching layer must guarantee that data produced by a confirmed write is observed by all subsequent reads.

Figure 1 shows the number of reads per second that violate read-after-write consistency, after 20 cache instances (*instances*) recover from a 10-second and a 100-second failure, respectively. These results are obtained using a trace derived from Facebook's workload [5, 21]. This number peaks immediately after the caches recover. It drops as the application processes a write that deletes an entry which happens to be stale. The percentage of stale reads is 6% of total reads at the peak for a 100-second failure.

A distributed crash recovery protocol for caches should also recover the latest application's working set to provide the highest cache hit ratio. If the working set evolves during an instance's failure, then recovering an entry that is not accessed in the future (or is evicted) is wasted work.

We present Gemini, a distributed crash recovery protocol for persistent caches. Gemini preserves read-after-write consistency in the presence of instance failures, reducing the number of stale reads of Figure 1 to zero. Gemini partitions an application's key space into fragments with one or more fragments assigned to an instance. When an instance fails, Gemini assigns its fragments to other instances. Gemini maintains a dirty list for each fragment of the failed instance. These dirty lists are used to recover a consistent state of the cache entries of the failed instance once it recovers. Moreover, Gemini transfers the latest working set to the recovering instance.

Technical challenges addressed by Gemini are as follows:

1. How to detect undesirable race conditions that may cause a cache to generate stale values? See Section 3.2.

2. How to restore the cache hit ratio of a recovering instance as fast as possible? See Section 3.2.2.

3. How does Gemini discard millions and billions of cache entries assigned to a fragment when it detects it cannot recover a consistent state of its keys? See Section 3.2.4.

4. How does Gemini provide read-after-write consistency with an arbitrary combination of its client and instance failures? See Section 3.3.

The primary contribution of this study is an efficient design of Gemini that addresses the above challenges. Gemini uses the eviction policy of an instance to discard its potentially invalid cache entries lazily. Its client detects stale cache entries and deletes them using a simple counter mechanism [13]. This mechanism facilitates read-after-write consistency in the presence of instance failures. Gemini uses leases [14] to prevent undesirable race conditions.

We evaluate Gemini using the YCSB benchmark [7] and a synthetic trace derived from Facebook's workload [5]. We measure the overhead of maintaining dirty lists of cache entries during an instance's failure and show it is insignificant. We compare Gemini's performance during recovery with a technique that discards the content of the cache. Our experiments show that Gemini restores the cache hit ratio by more than two orders of magnitude faster than a volatile cache.

2 Overview

Figure 2 shows Gemini's architecture, consisting of cache servers, a coordinator, and a client library. Table 1 provides a definition of these components and the terminology used in this paper. A cache server hosts one or more instances that store cache entries persistently. Each instance is assigned a fraction of its hosting cache server's memory. Cache entries are partitioned into subsets. Each subset is termed a *fragment*. Several fragments are assigned to an instance.

An application uses a Gemini client to issue requests to an instance. This client caches a *configuration* and uses it to route a request to a fragment for processing. A configuration is an assignment of fragments to instances. A Gemini client uses a deterministic function to map a request to an instance using a configuration. To illustrate, Figure 3 shows the processing of a read referencing a key K_i. A Gemini client uses a hash function to map this key to a cell of the configuration. This cell identifies a fragment. Its metadata identifies the fragment's mode and the instance that holds a lease on it.

A read that observes a cache miss queries the data store, computes a cache entry, and inserts this entry in an instance for future references. There are several policies for processing writes. With write-around [26], a write invalidates the relevant cache entry and updates the data store. With either write-through or write-back, the write updates the cache entry and the data store. While write-through applies writes to the data store synchronously, write-back buffers them and applies them to the data store asynchronously.

For the rest of this paper and due to lack of space, we make the following assumptions. First, Gemini and its application use the write-around policy. Gemini's implementation with

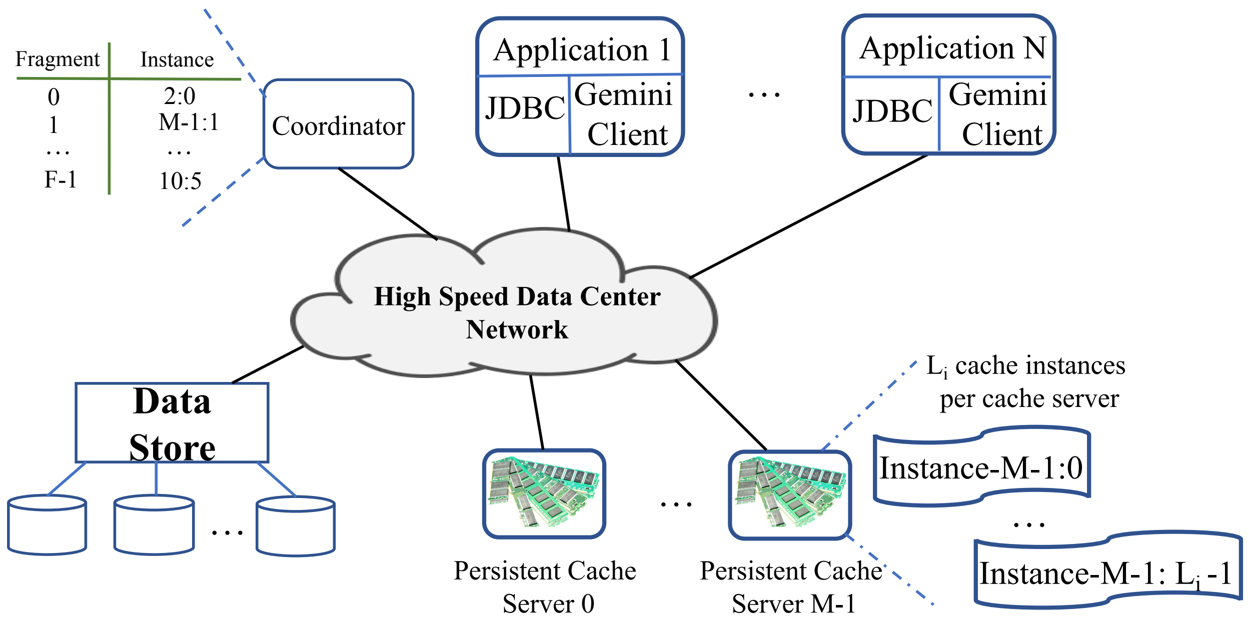


Figure 2: Gemini architecture.

Table 1: Terms and their definitions.

Term	Definition
Instance	A process for storing and retrieving cache entries persistently.
Fragment	A subset of cache entries assigned to an instance.
Configuration	An assignment of fragments to instances.
Configuration Id	A monotonically increasing integer variable that identifies each configuration published by the coordinator.
Coordinator	Grants leases on fragments to instances, maintains a configuration and publishes it.
Client	A software library that caches a configuration and is used by application to issue requests to an instance.
Primary replica $PR_{j,p}$	The replica of Fragment j in normal mode hosted on an instance I_p.
Secondary replica $SR_{j,s}$	The replica of Fragment j in transient and recovery modes hosted on an instance I_s.
Dirty list D_j	A list of keys deleted and updated from Fragment j stored in the instance hosting $SR_{j,s}$.

write-through is different. Second, we present a client side implementation of Gemini. Its server side implementation using its instances would be different.

2.1 Coordinator and Configuration Management

An instance must have a valid lease on a fragment in order to process a request that references this fragment. The instance obtains the lease from the *coordinator*. The lease has a fixed lifetime and the instance must renew its lease to continue processing requests that reference this fragment. When the coordinator assigns a lease on a fragment, say Fragment

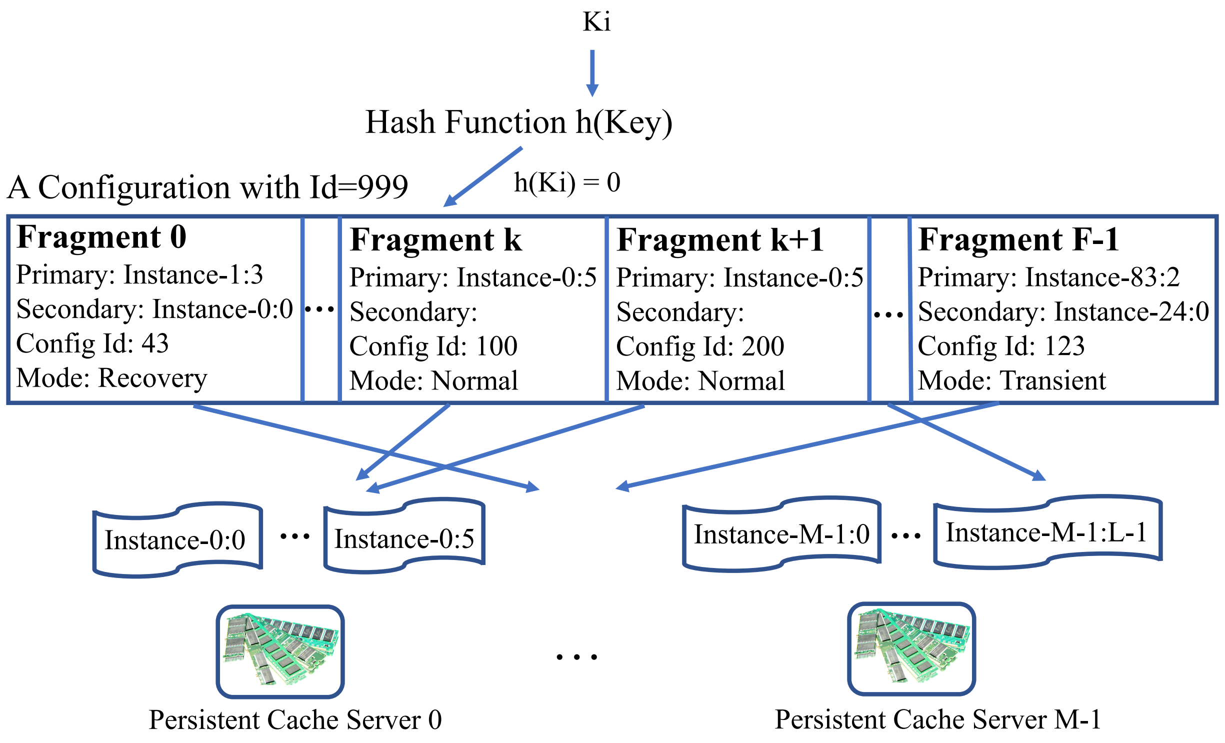


Figure 3: Client's processing of a request for K_i using a configuration.

j where j is a cell in Figure 3, it identifies the instance hosting either the fragment's *primary* replica $PR_{j,p}$ or its *secondary* replica $SR_{j,s}$. The first time Fragment j is assigned to an instance I_p, that replica is identified as its primary $PR_{j,p}$. Failure of this instance causes the coordinator to assign another instance I_s to host Fragment j, termed its secondary replica $SR_{j,s}$.

The coordinator maintains the configuration and grants leases. It identifies each of its published configurations with an increasing id. Each time an instance fails or recovers, the coordinator (a) computes a new configuration and increments the id, (b) notifies impacted instances that are available of the new id, and (c) inserts the new configuration as a cache entry in these instances. Instances memoize the latest configuration id. However, they are not required to maintain the latest configuration permanently because their cache replacement policy may evict the configuration.

To tolerate coordinator failures, Gemini's coordinator consists of one master and one or more shadow coordinators maintained using Zookeeper [16]. When the coordinator fails, one of the shadow coordinators is promoted to manage the configuration, similarly to RAM-Cloud [27].

2.2 Life of a fragment

Gemini manages a fragment in three distinct modes: normal, transient, and recovery, see Figure 4. Gemini assigns a fragment to an available instance in the cluster, constructing its primary replica. In normal mode, clients issue read and write requests to a fragment's primary replica.

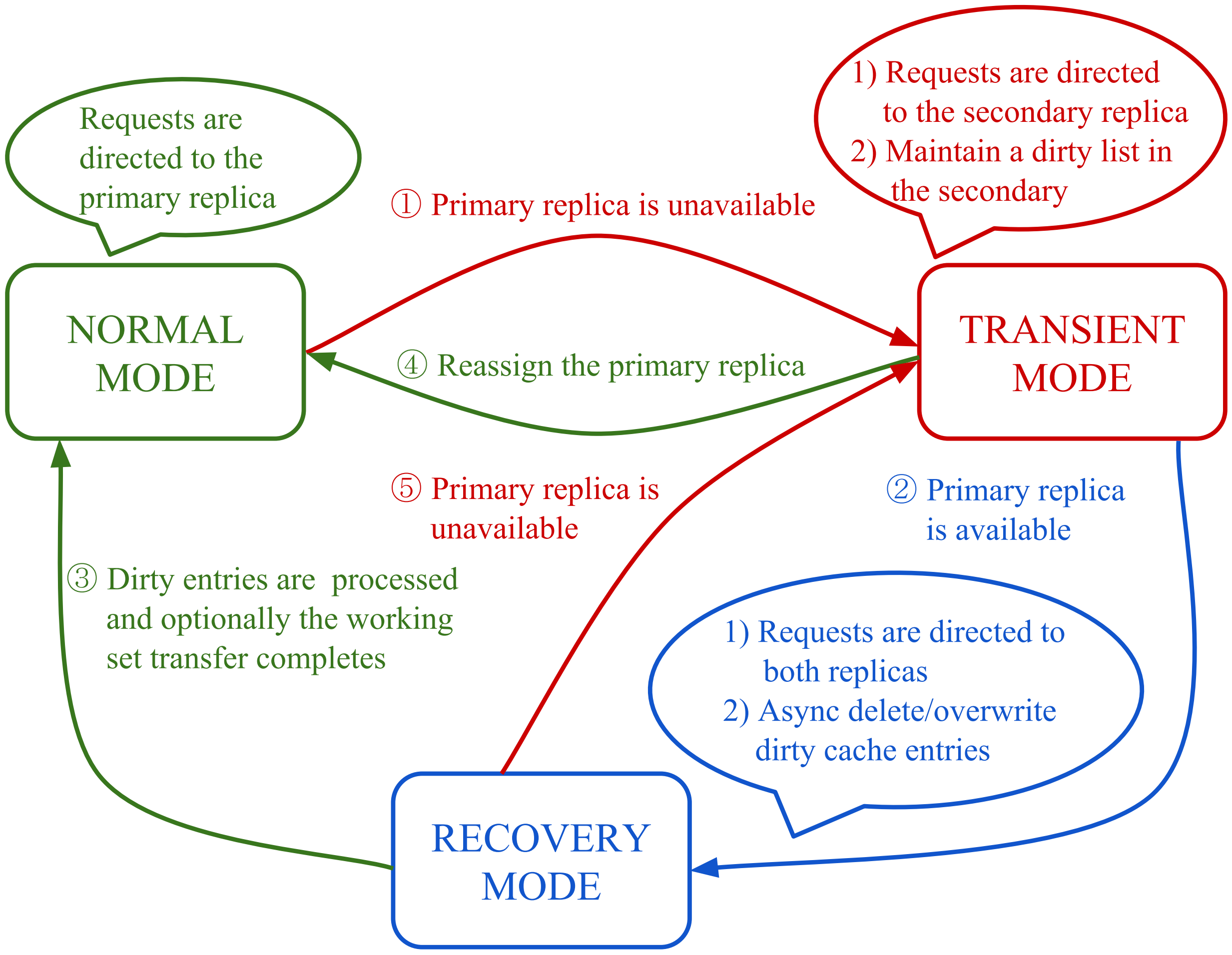

Requests are directed to the primary replica
1) Requests are directed to the secondary replica
2) Maintain a dirty list in the secondary
① Primary replica is unavailable
NORMAL MODE
TRANSIENT MODE
④ Reassign the primary replica
⑤ Primary replica is unavailable
② Primary replica is available
③ Dirty entries are processed and optionally the working set transfer completes
1) Requests are directed to both replicas
2) Async delete/overwrite dirty cache entries
RECOVERY MODE

Figure 4: State transition diagram of a fragment.

The primary replica becomes unavailable when its hosting instance fails. The coordinator assigns another instance to host this fragment which creates a secondary replica for the fragment. Moreover, Gemini transitions mode of the fragment to transient ①. Initially, read requests observe cache misses in the secondary replica since it is empty. These requests populate this replica similar to the normal mode of operation. A write request (a) invalidates the impacted cache entry in the secondary replica, and (b) maintains the impacted key in a dirty list. The dirty list is stored in the secondary replica.

The primary replica of a fragment becomes available once its instance recovers. The coordinator sets the fragment's mode to recovery ② and publishes a new configuration. This causes a Gemini client to fetch the dirty list of the fragment from the instance hosting its secondary replica. With a read or a write that references a key in this list, a client deletes the key from the primary replica. Moreover, a client transfers the latest working set from the fragment's secondary replica to its primary replica. This restores the cache hit ratio of the recovering instance that host the fragment's primary replica. Gemini also employs recovery workers to speed up processing dirty lists of fragments in recovery mode. Once the dirty list is processed and optionally transfer of the working set completes, Gemini transitions the fragment to normal mode ③. Should the primary replica become unavailable before the recovery completes, the coordinator transitions the fragment back to transient mode ⑤.

The coordinator may discard cache entries in the primary replica ④ if (a) the overhead of maintaining dirty cache entries outweighs its benefit, or (b) the dirty list is unavailable because the instance hosting the secondary replica either failed or evicted the dirty list.

During the interval of time, from when the instance hosting a fragment's primary replica fails, to when the coordinator publishes its secondary replica, a Gemini client suspends processing of writes that reference the fragment. This preserves read-after-write consistency. Moreover, all reads are processed using the data store. Once a client obtains the latest configuration that identifies a secondary, reads and writes are resumed using the secondary replica of the fragment.

2.3 Leases

A Gemini's client acquires Inhibit (I) and Quarantine (Q) leases [14] on application's cache entries to ensure read-after-write consistency. Moreover, its recovery worker acquires Redlease [4] on a dirty list to prevent another recovery worker from processing the same dirty list. While I and Q leases may collide on the same cache entry, it is not possible for these leases to collide with a Redlease. Similarly, a Redlease may collide with another Redlease and not either an I or a Q lease.

IQ leases and Redlease have a lifetime in the order of milliseconds. Gemini's coordinator grants leases on fragments to instances. These leases have a lifetime in the order of seconds if not minutes depending on the reliability of an instance. Below, we describe IQ leases and Redlease in turn.

Gemini uses the concept of sessions. A *session* is an atomic operation that reads and writes one cache entry and issues one transaction to the data store. It obtains leases on cache entries from an instance. Table 2 shows the compatibility of I and Q leases. An instance grants an I lease on a key to a request that observes a cache miss for the value of this key. Once the requester computes the missing value and inserts it in the cache, its I lease must be valid in order for its insert to succeed. Otherwise, the instance ignores the inserted value.

Table 2: IQ Lease Compatibility.

Requested	Existing Lease	
Lease	I	Q
I	Back off	Back off
Q	Void I & grant Q	Grant Q

I leases are incompatible with one another. When multiple concurrent reads observe a cache miss for the same key, only one of them is granted the I lease. The others back off and look up the cache again. The read granted the I lease queries the data store and populates the cache with the missing entry. When other concurrent requests try again, they find this entry and consume it. The I lease prevents the thundering herd phenomenon of [26], where the data store is flooded with concurrent queries that are identical.

A write request must acquire a Q lease on its referenced cache entry prior to deleting it (write-around). A Q lease voids an existing I lease on the key. This prevents the race condition where a concurrent read would populate an instance with a stale value [14]. With write-around, Q leases are compatible because the impacted cache entry is deleted and the order in which two or more sessions delete this entry does not result in an undesirable race condition. After acquiring a Q lease, the session updates the data store, deletes the key, and releases its Q lease. When a Q lease times out, the instance deletes its associated cache entry.

A Redlease protects a dirty list, to provide mutual exclusion between recovery workers that might process this list. Once a Redlease on a dirty list is granted to a recovery worker, a request by another worker must back off and try again.

3 Gemini Recovery Protocol

Given a large number of instances, multiple instances may fail and recover independently and concurrently. Gemini's recovery focuses on impacted fragments. It processes each fragment independent of the others, facilitating recovery of two or more fragments at the same time. Gemini uses IQ leases described in Section 2.3 to process a request that references a fragment in normal mode. Below, we describe the processing of a request that references a fragment in either transient or recovery mode. Appendix A provides a formal proof of read-after-write consistency.

3.1 Transient Mode

Fragment j in transient mode consists of both a primary replica $PR_{j,p}$ on a failed instance I_p and a secondary replica $SR_{j,s}$ on an available instance I_s. Once a client obtains a configuration that identifies a fragment in transient mode, it retries its outstanding requests using the secondary replica and directs all future requests to this secondary. A client processes read requests in the same way as in normal mode and populates the secondary.

In transient mode, the instance I_s hosting $SR_{j,s}$ maintains a dirty list for the fragment in anticipation of the recovery of the primary $PR_{j,p}$. The dirty list is represented as a cache entry. A write request appends its referenced key to the dirty list. The dirty list may be lost due to the instance I_s failing or its cache replacement technique evicting the dirty list. Either causes the coordinator to terminate transient mode and discard the primary replica $PR_{j,p}$. Next, the coordinator either promotes the existing secondary to be the primary or identifies a new primary for the fragment on a new instance.

To detect dirty list evictions, the dirty list is initialized with a marker. The marker enables Gemini to detect the race condition where one client appends to the list while the instance I_s evicts the list. Even though the client succeeds in generating a new dirty list, it is detected as being partial because it lacks the marker.

3.2 Recovery Mode

Once the coordinator detects a failed instance has recovered, Gemini starts to recover its fragments' primary replicas while allowing them to take immediate ownership of their valid cache entries to process requests. It employs recovery workers to process dirty keys, expediting recovery of a fragment. Recovery of a fragment is configurable in two ways. First, Gemini's recovery workers may either invalidate a dirty key from its fragment's primary replica or overwrite its value with the latest from the secondary replica. Second, one may enable or disable Gemini's transfer of the working set. This results in the four possible variations of Gemini shown in Figure 5. Below, we provide its details.

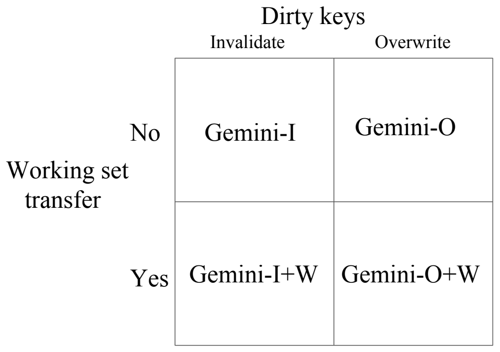


Figure 5: Four Gemini variations.

3.2.1 Processing Client Requests

When an instance becomes available, it is possible that only a subset of its fragments' primary replicas is recoverable. Those that lack dirty lists must be discarded as detailed in Section 3.2.4. Gemini uses Algorithm 1 to allow a primary replica to take immediate ownership of its still valid cache entries to process requests. Details of the algorithm are as follows. A client fetches the dirty list of each fragment in recovery mode from the instance hosting its secondary replica. A write request deletes its referenced key from both the primary and the secondary replicas. A read request checks if its referenced key is on the dirty list. If so, it deletes this key from the primary replica and acquires an I lease on this key from the instance hosting the primary replica. It looks up the key's value in the secondary replica. If found then it proceeds to insert the obtained value in the primary replica. Otherwise, it uses the data store to compute the missing value and inserts it in the primary.

3.2.2 Working Set Transfer

The application's working set may have evolved during an instance's failure. This means some of its cache entries are obsolete. To minimize cache misses, a Gemini client copies popular cache entries from a secondary to a primary. Gemini uses IQ leases to prevent race conditions. This is realized as follows, see lines 11 to 15 in Algorithm 1. When a request observes a cache miss in a primary, the client looks up its secondary. If it finds a value in the secondary, it inserts the entry in the primary. Otherwise, it reports a cache miss. A write request also deletes its referenced key in the secondary replica, see line 3 in Algorithm 2.

Gemini terminates the working set transfer once either (a) the cache hit ratio of the primary $PR_{j,p}$ exceeds a threshold h or (b) the cache miss ratio of the secondary $SR_{j,s}$ exceeds a threshold m. Benefits and costs of the working set transfer are quantified using h

Algorithm 1: Client: processing get on a key k mapped to a fragment j in recovery mode.

```
   Input: key $k$
   Result: key $k$'s value
   Let $PR_{j,p}$ be fragment $j$'s primary replica on instance $I_p$.
   Let $SR_{j,s}$ be fragment $j$'s secondary replica on instance $I_s$.
   Let $D_j$ be fragment $j$'s dirty list.
1  if $k \notin D_j$ then
2      v = $PR_{j,p}$.iqget($k$)                              // Look up the primary.
3      if v $\neq$ null then
4          return v                                          // Cache hit.
5      end
6  else
7      $PR_{j,p}$.iset($k$)               // Delete $k$ and acquire an I lease in the primary.
8      $D_j = D_j - k$
9  end
   /* Cache miss in the primary.                                                 */
10 if working set transfer is enabled then
11     v = $SR_{j,s}$.get($k$)                              // Look up the secondary.
12     if v $\neq$ null then
13         $PR_{j,p}$.iqset($k$, v)                          // Insert in the primary.
14         return v
15     end
16 end
   /* Cache miss in both replicas.                                               */
17 v = query the data store
18 $PR_{j,p}$.iqset($k$, v)
19 return v
```

Algorithm 2: Client: processing write on a key k mapped to a fragment j in recovery mode.

```
   Input: key $k$
   Let $PR_{j,p}$ be fragment $j$'s primary replica on instance $I_p$.
   Let $SR_{j,s}$ be fragment $j$'s secondary replica on instance $I_s$.
   Let $D_j$ be fragment $j$'s dirty list.
1  $PR_{j,p}$.qareg($k$)                        // Acquire a Q lease in the primary.
2  if working set transfer is enabled then
3      $SR_{j,s}$.delete($k$)                          // Delete $k$ in the secondary.
4  end
5  Update the data store
6  $PR_{j,p}$.dar($k$)                 // Delete $k$ and release the Q lease in the primary.
```

and m, respectively. We suggest to set h to be the cache hit ratio of the primary $PR_{j,p}$ prior to its instance failing while considering some degree of variation ϵ. We set m to be 1-h+ϵ.

Algorithm 3: Recovery worker: recover fragments in recovery mode.

```
   Input: A list of fragments in recovery mode
1  for each fragment $j$ in recovery mode do
2      Let $PR_{j,p}$ be fragment $j$'s primary replica on instance $I_p$.
3      Let $SR_{j,s}$ be fragment $j$'s secondary replica on instance $I_s$.
4      Let $D_j$ be fragment $j$'s dirty list.
5      Acquire a Redlease on $D_j$ in $SR_{j,s}$.
6      if fail to acquire the lease then
7          continue
8      end
9      if overwrite dirty keys is enabled then
10         for each key $k$ in $D_j$ do
11             $PR_{j,p}$.iset($k$)
12             v = $SR_{j,s}$.get($k$)
13             if v $\neq$ null then
14                 $PR_{j,p}$.iqset($k$, v)
15             else
16                 $PR_{j,p}$.idelete($k$)                // Release the I lease.
17             end
18         end
19     else
20         Delete keys in $D_j$ in $PR_{j,p}$
21     end
22     Delete $D_j$ and release the Redlease
23 end
```

3.2.3 Recovery Workers

Gemini uses stateless recovery workers to speed up recovery. These workers overwrite a cache entry on the dirty list in the primary with its latest value from the secondary. Algorithm 3 shows the pseudo-code of a recovery worker. A worker obtains a Redlease [4] on the dirty list of a fragment. All other lease requesters must back off and try again. This ensures different workers apply dirty lists on primary replicas of different fragments: one worker per fragment with a dirty list. Next, the worker fetches the dirty list from the secondary. For each key on this list, it deletes it from the primary and obtains an I lease on the key from its instance. Next, it looks up the key in the secondary. If it finds the key then it writes it to the primary. Otherwise, it releases its I lease and proceeds to the next key. Once all dirty keys are processed, the recovery worker deletes the dirty list and releases its Redlease.

Algorithm 3 also shows a setting that requires the recovery worker to delete dirty keys from the primary, see line 20. This is appropriate when the working set of an application evolves. In this case, overwriting keys using values from a secondary imposes an additional overhead without providing a benefit. Hence, deleting dirty keys is more appropriate.

Once a recovery worker exhausts the dirty list for a fragment, it notifies the coordinator to change the mode of this fragment to normal and publish a new configuration. This causes the clients to stop looking up keys in the dirty list of this fragment and to discard this dirty list.

3.2.4 Discarding Fragments

Gemini uses the coordinator's assigned configuration id to distinguish cache entries that can be reused from those that must be discarded. Each cache entry stores the configuration id that wrote its value. Moreover, each fragment of a configuration identifies the id of the configuration that updated it. For a cache entry to be valid, its local configuration id must be equal to or greater than its fragment's configuration id. Otherwise, it is obsolete and discarded. To recover a fragment's primary replica, its configuration id is restored to the value at the time of its instance failure. Interested readers may refer to Rejig [13] for a more detailed description and a proof of the protocol.

Example 3.1. Assume the coordinator's current configuration id is 999. Figure 3 shows Fragment F_k was assigned to Instance-0:5 (hosted on cache server 0) in configuration 100. Fragment F_{k+1} was assigned to the same instance in configuration 200.

When Instance-0:5 fails, the coordinator transitions its fragments to transient mode. Assume it assigns Instance-1:1 to host the secondary replicas of F_k and F_{k+1}. This results in a new configuration with id 1000. The id of F_k and F_{k+1} is set to 1000 because their assignment changed in this configuration. It also notifies Instance-1:1 of the new id and inserts the configuration as a cache entry in this instance.

Once Instance-0:5 recovers, the coordinator checks for the dirty lists of F_k and F_{k+1} in their secondary replicas on Instance-1:1. Assume the dirty list for F_k exists while that of F_{k+1} was evicted and is lost. Hence, the coordinator transitions F_k to recovery mode, sets F_k's configuration id to 100, its configuration id to 1001, and F_{k+1}'s configuration id to 1001. The last setting causes the cache entries of F_{k+1}'s primary replica on Instance-0:5 to be discarded. This is because all these entries are labeled with configuration ids no greater than 999 and this is smaller than the id of their fragment.

3.3 Fault Tolerance

Gemini's design tolerates arbitrary failures of its clients, recovery workers, and instances. We describe each in turn.

When a client fails during processing read or write requests, its outstanding I and Q leases expire after some time to allow other clients to process requests referencing these keys. When a client recovers from a failure, it fetches the latest configuration from an instance. If the instance does not have the configuration then the client contacts the coordinator for the latest configuration. Once it has the configuration then it proceeds to service requests.

When a recovery worker fails during recovering a fragment's primary replica, other recovery workers will later work on this fragment when its Redlease expires. It guarantees read-after-write consistency since deleting or overwriting a dirty key is idempotent.

An instance hosting a fragment's secondary replica may fail. If it fails while the fragment's primary replica is unavailable, the coordinator transitions the fragment to normal mode and discards its primary replica by changing its configuration id to be the latest id. If it fails before the recovery completes, clients terminate the working set transfer and recovery workers delete remaining dirty cache entries in the fragment's primary replica.

4 Implementation

We implemented a prototype of Gemini. Its client library is implemented with 3,000 lines of Java code. The coordinator is based on Google RPC [18] and implemented with 1,500 lines of Java code.[1] The coordinator distributes fragments owned by a failed instance uniformly across the remaining instances. When a failed instance recovers, the coordinator assigns these fragments back to the recovering instance. A key is mapped to a fragment by hash(key) % the number of fragments.

We implemented an instance on top of IQ-Twemcached, the Twitter extended version of memcached with the IQ framework. We extended IQ-Twemcached with only 40 lines of C code to maintain the configuration id described in Section 2 and Section 3. As Gemini's recovery protocol is agnostic to the underlying persistent storage media, we emulate a persistent cache using DRAM and an instance failure using Gemini's coordinator by removing it from the latest configuration. We verified our implementation is free from atomicity, consistency, and serializability violations using Polygraph [3].

5 Evaluation

Our evaluation answers two questions: *1) How fast can Gemini restore the system performance when an instance becomes available after a failure? 2) How effective is the working set transfer in restoring cache hit ratio of a recovering instance?*

As Gemini focuses on reusing still valid cache entries in a fragment's primary replica, the remaining number of valid cache entries is crucial to its performance. Many factors impact the number of dirty cache entries in a primary replica when its hosting instance fails. The application workload (read/write ratio) and the access pattern (probability of referencing a key) determine the probability that a key becomes dirty during its unavailability. The number of dirty keys is then dictated by the failure duration (seconds), the frequency of writes, and the system load. These are the parameters of our evaluation.

We evaluate Gemini in two modes: transient mode and recovery mode. Transient mode quantifies the overhead of maintaining dirty lists while recovery mode quantifies both the overheads and benefits of recovering a failed instance. Since Gemini recovers each fragment independently, this section quantifies costs and benefits for a single instance failure.

We compare Gemini with two baseline systems:

1. **VolatileCache:** Discard the content of an instance after recovering from a power failure. This approach removes the persistence property, simulating a volatile cache.

[1]This implementation lacks shadow coordinators and Zookeeper [16].

2. **StaleCache:** Use the content of an instance without recovering the state of stale cache entries or adjusting for an evolving access pattern. This technique produces stale data as in Figure 1.

We provide a comprehensive comparison of Gemini by exercising its possible configuration settings, see Figure 5. We present evaluation results using a synthetic trace derived from Facebook's workload [5] and YCSB [7]. Main lessons are as follows:

- Gemini guarantees read-after-write consistency in the presence of instance failures.
- Gemini maximizes the cache hit ratio of a recovering instance immediately upon its recovery. With a static access pattern, Gemini realizes this as quick as StaleCache without producing stale data.
- Gemini restores the cache hit ratio at least two orders of magnitude faster than VolatileCache under various system loads and application workloads.
- The rate to restore cache hit ratio of a recovering instance depends on its failure duration, the application workload, the system load, and the access pattern.
- The working set transfer maximizes the cache hit ratio of a recovering instance for workloads with access patterns that evolve quickly.

5.1 Synthetic Facebook-like Workload

This section uses statistical models for key-size, value-size, and inter-arrival time between requests for Facebook's workload [5] to evaluate Gemini. The mean key-size is 36 bytes, the mean value-size is 329 bytes, and the mean inter-arrival time is 19 μs. We generated a synthetic trace based on these distributions assuming a read-heavy workload (95% reads), a static working set (10 million records), a highly skewed access pattern (Zipfian with $\alpha = 100$), 5000 fragments with 50 fragments per instance, and a cache memory size equal to 50% of the database size [21].

Figure 6 shows cache hit ratio of a configuration consisting of 100 instances as a function of time. At the 50th second (x-axis), we fail 20 instances for 100 seconds. The cache hit ratio drops at the 50th second because the secondary replicas start empty. We observe that Gemini-O+W, StaleCache, and VolatileCache have a comparable cache hit ratio in normal and transient modes.

At the 150th second, the 20 failed instances are restarted, causing their fragments to enter recovery mode. Gemini-O+W restores its hit ratio immediately since it continues to consume still valid entries in recovering instances. StaleCache has a slightly higher hit ratio. However, it generates stale data, see Figure 1. VolatileCache has the lowest hit ratio due to the failed 20 instances losing their cache contents.

5.2 YCSB Workloads

We use YCSB [7] to evaluate Gemini's performance characteristics using an Emulab [31] cluster of 11 nodes: 1 server hosts a MongoDB (version 3.4.10) document store and a Gemini's coordinator. 5 servers each hosts one instance of IQ-Twemcached and 5 servers each hosts

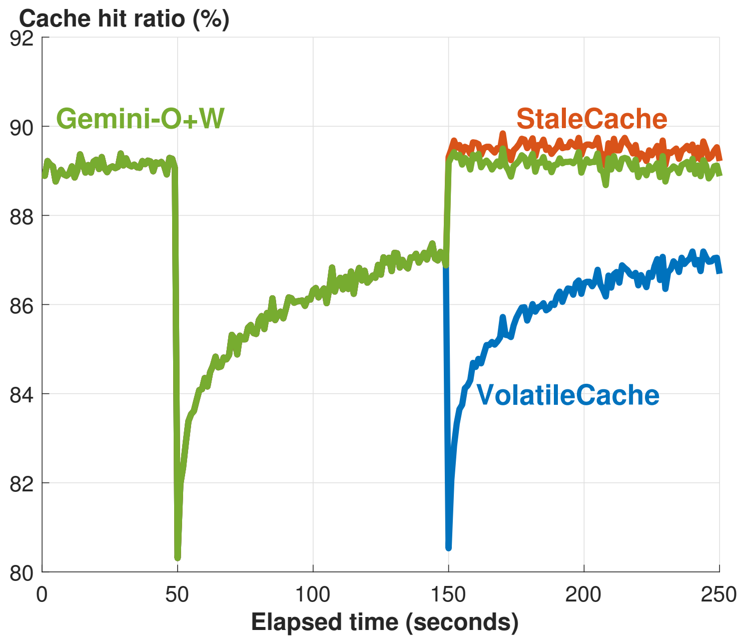


Figure 6: Cache hit ratio before, during, and after 20 instances fail for 100 seconds.

one YCSB client that generates a system workload. Each server is an off-the-shelf Dell Poweredge R430 with two 2.4 GHz 64-bit 8-Core, 64 GB memory, 200GB SSD and 1 Gbps networking card. These servers are connected using a 1 Gbps networking switch.

All experiments use a 10 million record YCSB database with a highly skewed Zipfian distribution with $\alpha = 100$. Each record is 1KB in size and is represented as a MongoDB document. The YCSB read command looks up a record in the caching layer first and, with misses, queries MongoDB to compute a cache entry, and populates the instance. The unique identifier of a YCSB record identifies a MongoDB document and the key of the corresponding cache entry. A YCSB update invalidates the cache entry and updates the corresponding document in MongoDB.

We construct 5000 fragments across the 5 instances, 1000 fragments per instance. An instance has sufficient memory to store all cache entries for its assigned fragments. We use Gemini's coordinator to emulate an instance failure by removing it from the latest configuration (without failing it). This causes each of the 1000 fragments of this instance to create a secondary replica, with 250 fragments' secondary replicas assigned to each of the remaining 4 instances. Clients are provided with the latest configuration, causing them to stop using the emulated failed instance. The content of the failed instance remains intact because its power is not disrupted. Similarly, we emulate an instance recovery by using the coordinator to inform the clients of the availability of the instance. Gemini recovers all fragments' primary replicas when the failed instance recovers. The working set transfer terminates when the cache hit ratio of the recovering instance is restored to the same value prior to its failure.

We consider both a low and a high system load. With a low system load, each client uses 8 YCSB threads to issue requests for a total of 40 YCSB threads (by 5 clients). With a high system load, the number of threads per client increases five-fold for a total of 200 YCSB threads issuing requests.

We consider YCSB workloads consisting of a mix of reads and updates: Workload A consists of 50% reads and 50% updates, Workload B consists of 95% reads and 5% updates. In several experiments, we also vary YCSB workload's percentage of updates from 1% to 10%, reducing its percentage of reads proportionally.

We also evaluate Gemini with workloads that have a static and an evolving access pattern. A static access pattern references data items based on a fixed distribution during the entire experiment. An evolving access pattern changes the access distribution during the instance's failure.

We report the mean of three consecutive runs with error bars showing the standard deviation. All the experiments described in the following sections are based on the YCSB workloads.

5.3 Transient Mode

Gemini imposes an overhead on a secondary replica by requiring it to maintain a dirty list. Each of the remaining 4 available instances maintains 250 dirty lists. Figure 7 shows the cache hit ratio of an instance, the overall system throughput, and the 90$^{\text{th}}$ percentile response time before, during, and after the instance's 10-second failure. The x-axis of this figure is the elapsed time. The first 10 seconds show system behaviors in normal mode. At the 10th second, we emulate the instance failure for 10 seconds. Reported results are with 1% writes. We observe similar results with YCSB's write-heavy workload A consisting of 50% writes.

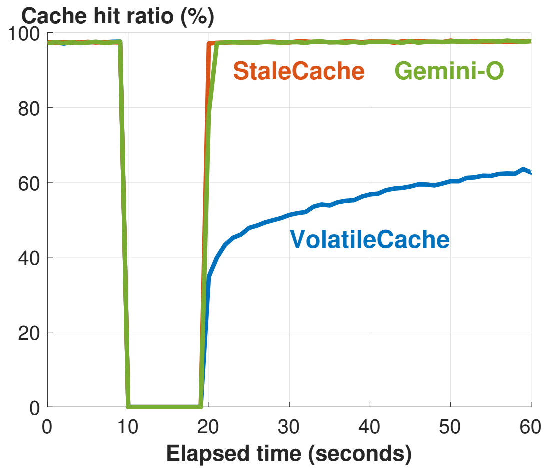
In transient mode, the failed instance does not process requests, providing a 0% cache hit ratio, see Figure 7.a. The overall throughput of the system in transient mode is identical between Gemini-O and its alternatives even though these alternatives generate no dirty lists. This is due to the time to apply the write to the data store is significantly higher, masking the overhead of appending the referenced key to the dirty list. Hence, there is no noticeable difference between Gemini and VolatileCache (or StaleCache) in transient mode.

5.4 Recovery Mode

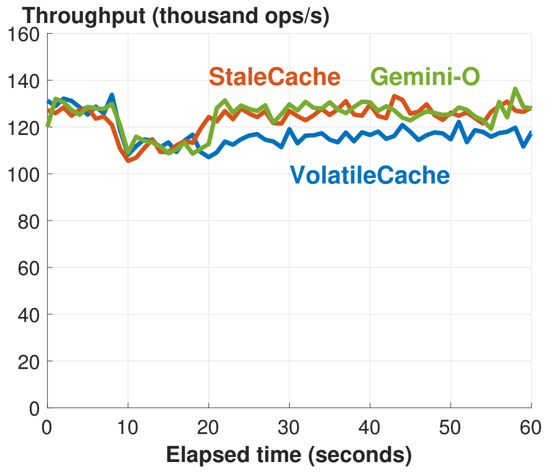
We start with workloads that have a static access pattern (workload-B) and focus on quantifying the impact of overwriting dirty keys with their latest values in secondary replicas (Gemini-O) and compare it with invalidating dirty keys (Gemini-I). Then, we evaluate Gemini with an access pattern that evolves during the instance's failure. We focus on quantifying the impact of working set transfer (Gemini-I+W) and compare it with Gemini-I.

5.4.1 Static Access Pattern

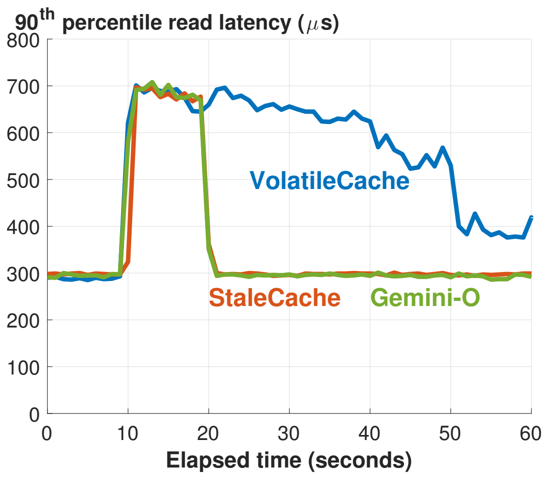
With Gemini-O+W, the recovery time includes the time to (a) overwrite dirty keys in a primary replica with their latest values in its secondary replica, and (b) restore cache hit ratio of the recovering instance. With a static access pattern, the first item dictates the



(a) Cache hit ratio of the failed instance.

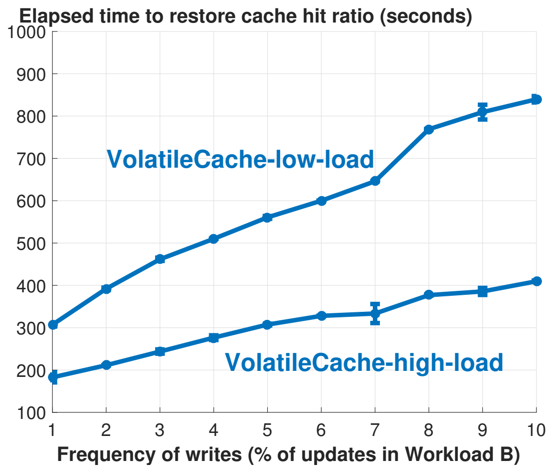


(b) Throughput.

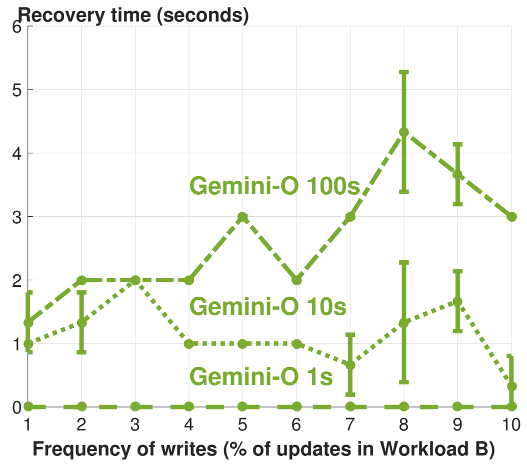


(c) 90th Percentile read latency.

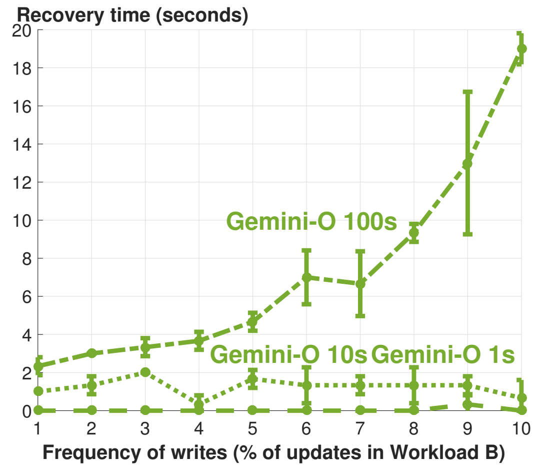
Figure 7: Performance before, during, and after a 10-second failure with a low system load and 1% update ratio.



(a) VolatileCache with a low and a high system load.



(b) Gemini-O with a low system load.



(c) Gemini-O with a high system load.

Figure 8: Elapsed time to (a) restore the recovering instance's cache hit ratio, (b-c) complete recovery.

recovery time because the access pattern to the persistent entries of the primary replica is unchanged.

Figure 8.a shows the time to restore cache hit ratio of the recovering instance with VolatileCache. Figures 8.b and 8.c show Gemini-O's recovery time as a function of the frequency of writes with a low and a high system load respectively. In these experiments, the recovery time with StaleCache is zero. However, it produces stale data of Figure 1. Gemini-O's recovery time is in the order of seconds with both a low and a high system load. With a low system load, the recovery workers overwrite most of the dirty keys. VolatileCache takes the longest because it deletes all cache entries of the recovering instance. The time to materialize these using the data store is in the order of hundreds of seconds. A higher system load materializes these faster by utilizing system resources fully.

The error bars in Figure 8.b show Gemini-O observes 1 to 2 seconds variation with a low system load. This is because it monitors cache hit ratio once every second. The recovery time is in the order of a few seconds and, to eliminate this variation, monitoring of cache hit ratio to terminate should become more frequent.

Figure 7.c shows the 90th percentile read latency for requests issued to all instances. This metric behaves the same with all techniques before and during the failure. After the failed instance recovers, StaleCache provides the best latency by restoring the cache hit ratio of the failed instance immediately while producing stale data. Gemini-O is slightly worse by guaranteeing consistency with negligible performance overhead compared to StaleCache. VolatileCache provides the worst latency because it must use the data store to populate the recovering instance.

The 90th percentile read latency of VolatileCache exhibits a long tail due to the skewed access pattern. Gemini-O reduces the average read latency by 20% and the 90th percentile read latency by 70%. With the 99th percentile read latency, the gap between VolatileCache and Gemini-O is only 19% since the cache hit ratio is 98% and the 99th percentile read latency is an approximate of the latency to query the data store for cache misses attributed to the write-around policy.

The throughput difference between VolatileCache and Gemini-O is less than 20% since the throughput is aggregated across 5 instances. With more instance failures, the throughput difference becomes more significant. We repeat the same experiment with Gemini-O+W and observe similar performance trends since the access pattern is static.

5.4.2 Transferring versus Invalidating Dirty Keys

Figure 9 shows Gemini's elapsed time to restore cache hit ratio of the recovering instance when it either deletes (Gemini-I) or overwrites (Gemini-O) dirty keys in the recovering instance. The x-axis of this figure varies the percentage of updates. The y-axis is the time to restore the cache hit ratio after a 100-second failure. We show results with both a high and a low system load.

Gemini-O is considerably faster than Gemini-I. Gemini-I observes cache misses for the dirty keys that it deletes, forcing a reference for it to query the data store. Gemini-O does not incur this overhead and restores the cache hit ratio faster.

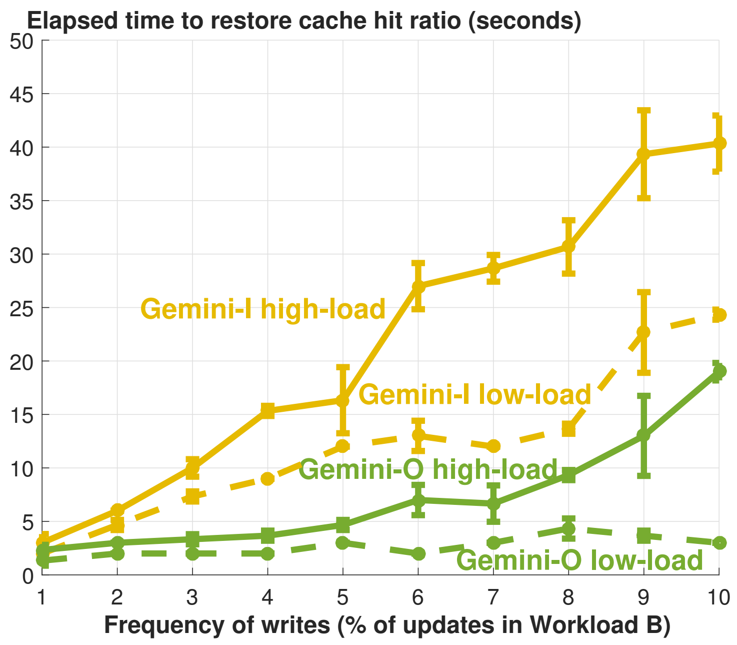


Figure 9: Elapsed time to restore the recovering instance's cache hit ratio with Gemini-I and Gemini-O after a 100-second failure with a low and a high system load.

5.4.3 Discarding Fragments

Lastly, we quantify the number of keys that are discarded due to failure of the instance hosting a fragment's secondary replica. We fail two instances (cache-1 and cache-2) one after another. We vary the total number of fragments as 10, 100 and 1000. Hence, each instance hosts 2, 20 and 200 fragments, respectively. Cache entries are distributed evenly across the fragments. The coordinator assigns fragments of a failed instance to other available instances in a round-robin manner. For example, with a total of 10 fragments, a fragment contains 1 million keys. When cache-1 fails, its first fragment F_0 is assigned to cache-2 and its second fragment F_1 is assigned to cache-3. When cache-2 fails before cache-1 recovers, the fragment F_0 must be discarded. The coordinator detects this and updates the fragment F_0's configuration id to the latest configuration id. This causes all clients to discard hits for those 1 million cache entries with a lower configuration id. With 100 and 1000 fragments, a maximum of 500,000 keys are discarded due to the failure of cache-2. The theoretical limit on this maximum is: $\lceil \frac{f}{n \times (n-1)} \rceil \times c$, where f is the total number of fragments, n is the number of instances, and c is the number of cache entries assigned to a fragment.

The second column of Table 3 shows the number of discarded keys in practice. This number is lower than the theoretical maximum because a write may delete an entry that must be discarded so that a future read observes a cache miss on this key. Results of Table 2 are with a high system load and 1% update ratio.

Table 3: Gemini's number of discarded keys with respect to the total number of fragments.

Total number of fragments	Number of discarded keys (mean ± standard deviation)	Maximum number of discarded keys
10	975,079 ± 29	1,000,000
100	487,374 ± 55	500,000
1000	487,397 ± 181	500,000

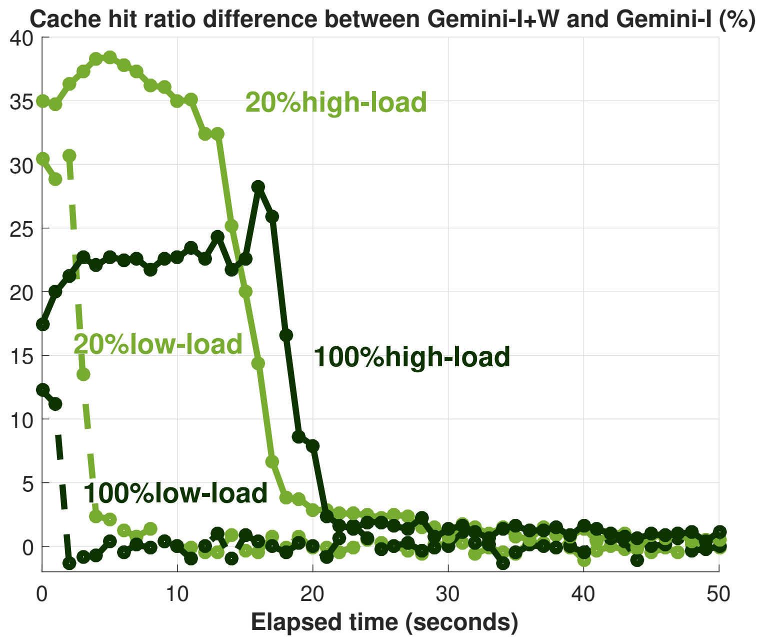


Figure 10: Cache hit ratio improvement by the working set transfer with a 20% and a 100% access pattern change, a low and a high system load.

5.4.4 Evolving Access Pattern

We emulate two evolving access patterns using the 10 million record YCSB database with Workload B as follows. We partition records into two sets, A and B, each with 5 million records. Prior to the instance's failure, all references are directed to records in A, never referencing those in B. After the failure, we direct all references to records in B with the same distribution as to that in A. This results in a 100% change in the access pattern. Alternatively, for a 20% change in the access pattern, we switch the most frequently accessed 1 million records in A with those in B.

A failure triggers the switch from records in A to those in B. This causes a different set of records to be materialized in the secondary replicas as compared to those persisted in the primary. The workload remains unchanged once the failed instance recovers.

Figure 10 shows the cache hit ratio difference between Gemini-I+W and Gemini-I after the failed instance recovers with both a low and a high system load. A higher difference demonstrates the superiority of Gemini-I+W. The difference is significant and lasts a few seconds with a low system load. The difference lasts longer with a high system load because (a) these requests transfer the working set to the recovering instance with Gemini-I+W, and (b) these requests observe cache hits on the transferred entries. In contrast, Gemini-I must query the data store for the new working set. This is considerably slower than Gemini-I+W fetching the working set from a secondary replica.

5.5 Gemini's Worst Case Scenario

Theoretically, if the entire working set changes during an instance's failure, Gemini incurs two overheads during recovery that provide no benefit. First, recovery workers overwrite dirty keys in the recovering instance but these keys will never be referenced. Second, for every read that observes a miss from the recovering instance, the lookup in an instance hosting the fragment's secondary replica also reports a miss because the access pattern changed. Duration of the first is dictated by the number of dirty keys. Duration of the second is dictated by the termination condition of working set transfer.

We quantify the overhead of Gemini by changing the working set completely before and after the failed instance recovers. The experiment issues a high load and generates 810,076 dirty keys during the instance's failure.

The average read latency increases by 10% due to the additional lookup on a secondary replica. The average update latency increases by 21% since it must be processed in both the secondary and the primary replicas. Each client also consumes significantly more CPU resources (50% more) to recover the instance though future read requests never reference these keys again. The recovery time lasts 70 seconds.

6 Related Work

Gemini is inspired by existing work on persistent caches [29, 11, 30, 33, 32, 24, 19], distributed crash recovery protocols [28, 22, 27], the working set transfer [20, 35, 17, 26, 6], and configuration managements [15, 10, 1]. Gemini uses IQ-framework [14] to ensure read-after-write consistency and Redlease [4] to maximize efficiency of recovery workers.

6.1 Persistent Caches

NAND flash-based key-value caches provide higher storage density, lower power per GB and higher GB per dollar than DRAM. RIPQ [29], Facebook's photo cache, reduces write amplification by grouping similar priority keys on flash. Facebook McDipper [11] and Twitter FatCache [30] leverage flash as a larger storage medium to cache larger working sets. Bluecache [33] is also a flash-based cache that uses hardware accelerators to enhance performance and reduce cost. NVMcached [32, 24] is a persistent memcached redesigned for NVM to reduce the number of CPU cache line flushes and minimize data loss with specially designed data structures. These work focus on the performance and cost-savings with persistent caches. Gemini focuses on resolving stale cache entries in a recovering instance that were updated during its failure.

6.2 Distributed Crash Recovery

RAMCloud [27] is a DRAM-based key-value data store that recovers gigabytes of data within seconds. RAMCloud maintains only one copy of data in DRAM and relies on fast crash recovery to provide high availability. Gemini is different since it can choose to not recover the cached content of a failed instance since the data is still available in the backend data store.

Two-phase commit [28] based systems (e.g., Sinfonia [2]) and Paxos [22] based systems (e.g., Spanner [8]) ensure consistency in the event of node or network failures. Gemini focuses on reusing the still valid content in a recovering instance after a failure, but not maintaining consistency across multiple replicas.

Facebook's memcached cluster [26] employs a component named mcrouter that routes a client request to a cache instance. It minimizes inconsistency by guaranteeing eventual delivery of an invalidation message. A mcrouter logs an invalidation message to its local disk when it fails to deliver the message. Later, it retries the delivery of this message until success. Invalidation messages of a failed instance are spread across many mcrouters. Gemini uses its clients to record a list of dirty keys for each fragment hosted on a failed instance. A dirty list is represented as a cache entry in its secondary replica. A Gemini client uses the dirty list to discard stale entries in the primary replica to guarantee read-after-write consistency.

6.3 Working Set Transfer

Key-Value storage systems, e.g., Rocksteady [20], provide on-demand live migration with low impact on the system performance by leveraging immediate transfer of ownership and load-adaptive replay. Gemini is different since its working set transfer is optional. Its objective is to enhance the cache hit ratio of a recovering instance.

Zhu et al. [35] propose a migration technique that migrates the most popular data items from a server to be shut down to remaining servers when the system scales down. It also migrates the most popular data items from an existing server to warm up a new server when the system scales up. Hwang and Wood [17] describe a similar technique that uses load balancers to migrate data items when the configuration changes. Facebook [26] employs a cold cluster warmup system that allows clients to warm up a cold cluster with data from a warm cluster. These migration techniques suffer from undesirable race conditions that produce stale data. Gemini's working set transfer ensures read-after-write consistency and its purpose is to maximize a recovering instance's performance. Specifically, (a) Gemini overwrites dirty keys in a primary replica with their latest values from its secondary replica, (b) Gemini transfers the latest working set from the secondary replica to its primary replica.

Redis [6], a popular key-value cache, supports atomic migration of keys between two Redis instances. However, the migration blocks both instances. Gemini uses IQ leases to ensure read-after-write consistency while serving online requests.

6.4 Configuration Management

The design of Gemini's coordinator, configuration management and distribution are inspired by previous work, e.g., Google File System [15], Hyperdex [10], and Slicer [1].

Google File System (GFS) [15] is a distributed file system for distributed data-intensive applications. A GFS file consists of a list of chunks. Each chunk is associated with a chunk version number. The master maintains the latest chunk version number for each chunk and detects a stale chunk if its version number is lower than the master's chunk version number. Gemini associates keys and fragments with configuration ids and updates a fragment's configuration id to the latest to discard entries that it cannot recover.

Hyperdex [10] is a distributed key-value store. It employs a coordinator that manages its configurations 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 server and client cache the latest configuration. A client embeds its local configuration id on every request to a server and discovers a new configuration if the server's configuration id is greater. Gemini differs in that it stores the configuration id with a cache entry to discard entries it cannot recover.

Slicer [1] is Google's general purpose sharding service. It maintains assignments using generation numbers. Slicer employs leases to ensure that a key is assigned to one slicelet (equivalent to an instance) at a time. Applications are unavailable for a maximum 4 seconds during an assignment change due to updating leases to reflect the latest generation number (and assignment) to slicelets. Gemini is different in several ways. First, while its instances may cache a copy of configuration (Slicer's assignment), they do not use it to decide whether to process a client request or not, it uses its local copy of configuration id for this purpose. Second, Slicer is agnostic to applications which compromises read-after-write consistency with assignment changes. Gemini associates configuration id with keys and fragments to detect and discard cache entries that it cannot recover.

7 Conclusion and Future Work

Today's state-of-the-art cache instances lose their content after a power failure because their DRAM is volatile. With a persistent cache, while the content of a failed instance remains intact, its cache entries may become stale if they are updated during the instance's failure.

Gemini creates a secondary replica for a fragment while the instance hosting its primary replica fails, enabling requests that reference the fragment to benefit from the high performance of the caching layer. It maintains a dirty list identifying those cache entries of the fragment that were impacted by application writes. Once the primary replica becomes available, Gemini either deletes its dirty keys or overwrites their values with those from the secondary replica. Moreover, Gemini transfers the latest working set to the primary replica.

We plan to extend Gemini's recovery protocol to support multiple replicas per fragment. Replication with persistent caches raises several design challenges. For example, as long as each replica has the latest value of a cache entry, is it important for all replicas of a fragment to be identical given that a fragment is a subset of data? If the answer to this question is affirmative then the next question is how to maintain replicas identical while performing cache evictions. One approach is for a master replica to broadcast its eviction decisions to slave replicas. Another possibility is to forward the sequence of requests referencing the master replica to the slave replicas. Assuming replicas use the same cache replacement technique, their eviction decisions should be identical given the same sequence of references. Trade-offs associated with these designs shape our immediate research directions.

Acknowledgments

We thank the anonymous reviewers of the ACM Middleware and Marc Shapiro for their valuable comments and helpful suggestions.

A Proof for Read-after-write Consistency

Without loss of generality, we prove the following theorems on Gemini-O+W using the write-around policy. The proof for the configuration management is presented in Rejig [13].

Theorem 1. *Gemini guarantees read-after-write consistency for processing requests on a key k mapped to a fragment j in its primary replica $PR_{j,p}$ hosted on an instance I_p in normal mode.*

It is trivial to see that concurrent writes are serialized at $PR_{j,p}$. Lemma 2 proves that Gemini guarantees read-after-write consistency with a read that observes a cache miss and a write referencing the same key k concurrently.

Lemma 2. *$PR_{j,p}$ serializes a read r that observes a cache miss and a write w on a key k concurrently.*

Proof. Let v be the initial value of k in the data store. (k, v) does not exist in $PR_{j,p}$ initially. $PR_{j,p}$ grants an I lease on k for r since it observes a cache miss. Then, r fetches its value from the data store and inserts it in $PR_{j,p}$ if the I lease is still present. w acquires a Q lease on k in $PR_{j,p}$, updates v to v' in the data store, atomically deletes k and releases the Q lease. There are two major cases to consider.
Case I: r's insertion happens before w's acquiring a Q lease. r succeeds in inserting (k, v) in $PR_{j,p}$ and returns v to the application. When w completes, the inserted (k, v) in $PR_{j,p}$ is deleted. r is serialized before w.
Case II: w's acquiring a Q lease happens before r's insertion. r's insertion will fail since the I lease is voided by the Q lease. If r queries the data store before w updates the data store, r returns v to the application and r is serialized before w. Otherwise, r returns v' to the application and w is serialized before r. ■

Theorem 3. *Gemini preserves read-after-write consistency for a key k mapped to a fragment j in recovery mode when its primary replica $PR_{j,p}$ becomes available.*

Theorem 1 proves that Gemini achieves read-after-write consistency for fragment j in normal mode. When $PR_{j,p}$ becomes available, a client processes reads using the data store before they fetch fragment j's dirty list D_j from f's secondary replica $SR_{j,s}$. Lemma-4 and Lemma-5 prove that Gemini preserves read-after-write consistency after fetching D_j with a read and a write request referencing the same key k concurrently. We can treat overwriting dirty keys in D_j performed by the recovery worker as read requests that reference keys in D_j which is also proven by Lemma-4 and Lemma-5.

Lemma 4. *Gemini preserves read-after-write consistency after clients fetching the dirty list D_j from the secondary $SR_{j,s}$ and k does not exist in the primary $PR_{j,p}$.*

Proof. The read request acquires an I lease and observes a miss on k in $PR_{j,p}$. There are two cases.
Case I: k does not exist in $SR_{j,s}$. The read request observes a miss in $SR_{j,s}$ and queries the data store. It is trivial to see that $PR_{j,p}$ serializes the concurrent read and write requests since I and Q leases are incompatible with each other.
Case II: k exists in $SR_{j,s}$. The read request reads k from $SR_{j,s}$ and inserts its value v into $PR_{j,p}$ if the I lease still exists. If the write request's acquiring a Q lease in $PR_{j,p}$ happens before the insert, the insert fails since the Q lease deletes the I lease associated with k. Otherwise, the insert succeeds and v is valid to consume. If releasing the Q lease in $PR_{j,p}$ happens before acquiring the I lease, k must have been deleted in $SR_{j,s}$ and Case I proves this case. ■

Lemma 5. *Gemini preserves read-after-write consistency after clients fetching the dirty list D_j from $SR_{j,s}$ and k exists in $PR_{j,p}$.*

Proof. There are two cases.
Case I: $k \in D_j$. The read request knows that k is dirty and treats it as a cache miss. Lemma-4 proves this case.
Case II: $k \notin D_j$. The read request observes a cache hit and consumes the value. ■

References

[1] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, Roberto Peon, Larry Kai, Alexander Shraer, Arif Merchant, and Kfir Lev-Ari. Slicer: Auto-Sharding for Datacenter Applications. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 739–753, Savannah, GA, 2016. USENIX Association.

[2] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: A New Paradigm for Building Scalable Distributed Systems. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 159–174, New York, NY, USA, 2007. ACM.

[3] Yazeed Alabdulkarim, Marwan Almaymoni, and Shahram Ghandeharizadeh. Polygraph: A Plug-n-Play Framework to Quantify Anomalies. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*, pages 1308–1311, 2018.

[4] S. Sanfilippo (antirez) and M. Kleppmann. Redlease and How To do distributed locking. http://redis.io/topics/distlock and http://martin.kleppmann.com/2016/02/08/how-to-do-distributed-locking.html, 2018.

[5] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. ACM.

[6] Redis contributors. Redis. https://redis.io/, 2018.

[7] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[8] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-Distributed Database. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 261–264, Hollywood, CA, 2012. USENIX Association.

[9] Peter J. Denning. The Working Set Model for Program Behavior. In *Proceedings of the First ACM Symposium on Operating System Principles*, SOSP '67, pages 15.1–15.12, New York, NY, USA, 1967. ACM.

[10] Robert Escriva, Bernard Wong, and Emin Gün Sirer. HyperDex: A Distributed, Searchable Key-value Store. *SIGCOMM Comput. Commun. Rev.*, 42(4):25–36, August 2012.

[11] Facebook. Mcdipper. `https://www.facebook.com/notes/facebook-engineering/mcdipper-a-key-value-cache-for-flash-storage/10151347090423920/`, 2018.

[12] Daniel Ford, François Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in Globally Distributed Storage Systems. In *Presented as part of the 9th USENIX Symposium on Operating Systems Design and Implementation*, Vancouver, BC, 2010. USENIX.

[13] Shahram Ghandeharizadeh, Marwan Almaymoni, and Haoyu Huang. Rejig: A Scalable Online Algorithm for Cache Server Configuration Changes. Technical Report 2018-05 `http://dblab.usc.edu/Users/papers/rejig.pdf`, USC Database Laboratory, 2018.

[14] Shahram Ghandeharizadeh, Jason Yap, and Hieu Nguyen. Strong Consistency in Cache Augmented SQL Systems. In *Proceedings of the 15th International Middleware Conference*, Middleware '14, pages 181–192, New York, NY, USA, 2014. ACM.

[15] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[16] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[17] Jinho Hwang and Timothy Wood. Adaptive Performance-Aware Distributed Memory Caching. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, pages 33–43, San Jose, CA, 2013. USENIX.

[18] Google Inc. Google Protocol Buffer. https://developers.google.com/protocol-buffers, 2018.

[19] Intel. pmem. http://pmem.io/, 2018.

[20] Chinmay Kulkarni, Aniraj Kesavan, Tian Zhang, Robert Ricci, and Ryan Stutsman. Rocksteady: Fast Migration for Low-latency In-memory Storage. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 390–405, New York, NY, USA, 2017. ACM.

[21] USC Database Laboratory. Facebook Workload Generator. https://github.com/scdblab/fbworkload/tree/middleware18, 2018.

[22] Leslie Lamport. The Part-time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[23] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. Existential Consistency: Measuring and Understanding Consistency at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 295–310, New York, NY, USA, 2015. ACM.

[24] Virendra J. Marathe, Margo Seltzer, Steve Byan, and Tim Harris. Persistent Memcached: Bringing Legacy Code to Byte-Addressable Persistent Memory. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, Santa Clara, CA, 2017. USENIX Association.

[25] memcached. memcached. `https://memcached.org/`, 2018.

[26] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, 2013. USENIX.

[27] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast Crash Recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 29–41, New York, NY, USA, 2011. ACM.

[28] D. Skeen and M. Stonebraker. A Formal Model of Crash Recovery in a Distributed System. *IEEE Trans. Softw. Eng.*, 9(3):219–228, May 1983.

[29] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. RIPQ: Advanced Photo Caching on Flash for Facebook. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 373–386, Berkeley, CA, USA, 2015. USENIX Association.

[30] Twitter. Fatcache. `https://github.com/twitter/fatcache`, 2018.

[31] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):255–270, December 2002.

[32] Xingbo Wu, Fan Ni, Li Zhang, Yandong Wang, Yufei Ren, Michel Hack, Zili Shao, and Song Jiang. NVMcached: An NVM-based Key-Value Cache. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '16, pages 18:1–18:7, New York, NY, USA, 2016. ACM.

[33] Shuotao Xu, Sungjin Lee, Sang-Woo Jun, Ming Liu, Jamey Hicks, and Arvind. Bluecache: A Scalable Distributed Flash-based Key-value Store. *Proc. VLDB Endow.*, 10(4):301–312, November 2016.

[34] Yiying Zhang, Gokul Soundararajan, Mark W. Storer, Lakshmi N. Bairavasundaram, Sethuraman Subbiah, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Warming Up Storage-Level Caches with Bonfire. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 59–72, San Jose, CA, 2013. USENIX.

[35] Timothy Zhu, Anshul Gandhi, Mor Harchol-Balter, and Michael A. Kozuch. Saving Cash by Using Less Cache. In *4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 12)*, Boston, MA, 2012. USENIX.