The Pennsylvania State University
The Graduate School

ADAPTING KEY-VALUE STORAGE SYSTEMS TO MINIMIZE COST IN THE PUBLIC CLOUD

A Thesis in
Computer Science and Engineering
by
Nader Alfares

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

May 2020

The thesis of Nader Alfares was reviewed and approved by the following:

Bhuvan Urgaonkar
Associate Professor of Computer Science and Engineering
Thesis Advisor

George Kesidis
Professor of Computer Science and Engineering
Thesis Co-Advisor

Viveck Cadambe
Assistant Professor of Electrical Engineering
Thesis Co-Advisor

Chita R. Das
Professor of Computer Science and Engineering
Head of The Computer Science and Engineering Department

Abstract

The growth of big data analytics and the volatility/diversity of pricing across public cloud services have presented many opportunities for optimization tailored to the need of an application. We focus our optimization on key-value storage systems in the public cloud. Such systems are widely popular due to their simplified semantics that allows applications to scale rapidly in multi-user environments, producing challenges on resource management and fairness guarantees. Since public cloud providers grant tenants to place their resources in specific regions (or datacenters), we divide our work into two settings of optimizations. First, we consider an optimization of a system within a single datacenter. Specifically, we deal with the caching layer that is often placed along with the database. We conduct our work by considering an object sharing framework to minimize storage cost. Second, we consider an optimization of a geographically distributed storage system that guarantees linearizability while meeting Service-Level Agreement (SLA). We rely on collected data for public cloud pricing models and performance measurements in our optimization formulation. In addition, we consider an optimistic approach and heuristic based optimizations.

Contents

List of Figures

List of Tables

Acknowledgments

> "To live is to risk it all. Otherwise, you're just an inert chunk of randomly assembled molecules drifting wherever the universe blows you"
> -Rick Sanchez

I wish to express my sincere appreciation to my advisors: Professor Bhuvan Urgaonkar, Professor George Kesidis and Professor Viveck Cadambe for their guidance and encouragement throughout my studies, as well as to my fellow researchers at the CSL lab in Penn State. I would also like to extend my sincere thanks to my family and friends for their continuous support.

Chapter 1
Introduction

The pricing volatility and diversity across public cloud services have presented many opportunities to provide better Quality of Service (QoS) while minimizing operational costs. As many businesses today are looking for better ways to migrate their applications to the public cloud, our focus is in the interest of applications that rely on key-value (KV) storage systems [1]. Essentially, such applications grant users to write into the database through PUT/SET commands or read from the database through GET commands while abstracting the managed resources. Similarly, such abstraction is provided to tenants when providing storage in the form of Software as a Service (SaaS) (e.g., S3 [2] and ElastiCache by AWS [3]). When unfolding the abstraction, performance, and fairness related guarantees are a challenging aspects of such systems [4–7].

In this thesis, we examine two standard settings for key-value store systems in the public cloud. The first setting considers optimization at the level of a single region (or a datacenter). In such a setting, we consider a KV storage system that consists of a single database, in-memory caching servers, and web servers (often referred to as proxy servers or edge servers). Our goal is to provide high caching performance (i.e., hit rates) in a multi-user caching system while maximizing memory utilization. Here, we consider object sharing as our main framework to achieve such objectives [8]. An example of such applications is a Content Distribution Network (CDN). CDNs use proxy servers to minimize the latency of retrieving data from a centralized database. Each server can serve a large population of end-users. We consider a system with J proxies that services database requests from users. Also, we consider a database with a size of N data objects and a caching unit with a size of B data objects (where $B \ll N$). A proxy server allocates a cache size of b objects,

where $b \leq B$. When an object is stored in the cache, read requests from proxy can be fulfilled by simply reading the data immediately from the cache storage; without sending a request to the centralized database. On a cache miss, the proxy sends a read request to the centralized database and updates the cache table to include the new object. In cases where the cache allocated for the proxy is full, the proxy performs an eviction to include the new object according to the policy upon agreement. Hence, eviction policies have a significant role in request latencies. One of the conventional policies used in caching systems is the Least-Recently-Used (LRU). In general, it allows for content with higher popularity to experience fewer evictions. Therefore, minimizing the number of cache misses. Given the setting described above, we implement such caching system with a modification on the LRU policy to minimize cost while preserving high performance.

Secondly, public providers often grant tenants to choose the regions in which they would allocate their resources. We consider a KV storage system that is geographically distributed. Such systems are naturally geo-distributed due to the need for tolerating failures while preserving strong consistency. Given such constraint, the goal is to construct a system that places replicated/erasure coded data across regions that minimizes the operational cost while preserving Service-Level Objective (SLO). We rely on the diversified pricing of services in the public cloud and the different workload properties to produce an adaptive system framework.

The thesis is constructed as the following. Chapter 2 focuses on the single region optimization; Chapter 3 on the multi-region optimization. The problem definition, background, related work, and evaluation are discussed separately for each system in their respective chapters.

Chapter 2
Optimization Within a Single Region

We consider a simple caching system model in the public cloud illustrated in Figure 2.1. The database and caching unit are, typically, located in the same datacenter. On the contrary, web servers can be on different sites (often referred to as proxy/edge servers). A web server sends a GET request for a key to the caching unit. Consequently, the caching unit responds with the value if found. Otherwise, it responds with a read miss. In case of a read miss, the web server forwards the GET request to the database. We focus our efforts to enhance the performance of such system (i.e., higher hit rates and resource utilization) for a multi-user environment; sharing the cache while respecting their individual Service-Level Agreement (SLA). Specifically, we consider a system with "paying" users for their amount of cache allocated. We implement our prototype by modifying Memcached [9] and employ object sharing to meet our objectives. We implement our prototype for object sharing based on [8] and evaluate its performance.

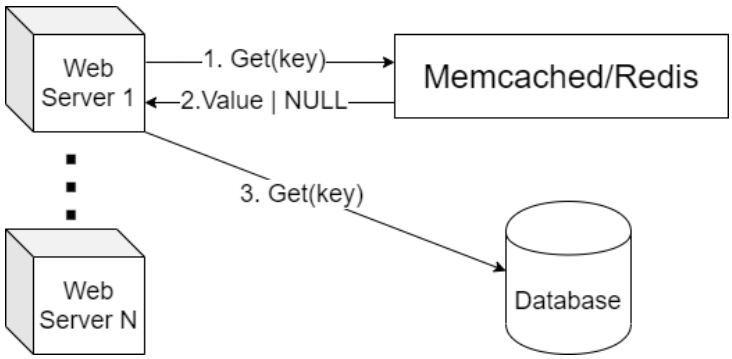


Figure 2.1: Simple model illustrating a caching system for a key-value store.

2.1 Background and Related Work

This section describes some elements of Memcached that are highly relevant to our work. Specifically, it describes the different data structures that Memcached utilizes to effectively store objects in-memory. In addition, it describes two forms of an LRU replacement policy that Memcached provides. Finally, it briefly describes the object sharing framework that we include in this work [8].

2.1.1 Memcached

Memcached (MCD) is an open source in-memory key-value store system that is used to improve the performance of database driven applications. Using MCD's API, clients can interact with the server to read and write from memory. On a read miss, the application layer is responsible for retrieving the value from the database and updating the caches accordingly. In distributed MCD servers, the system appears to end users as a single MCD instance. To implement such notion, each key is mapped to a single physical server using *consistent hashing* which is performed by the client at the application layer [10]. Accordingly, the MCD server updates its state before responding to the client. Updating the state of an MCD instance consists of updating LRUs and the value of the key for write requests.

2.1.1.1 Slab Allocation

MCD uses slab allocation as its primary scheme to store different objects with different sizes. Figure 2.2 illustrates the different units of storage in MCD. By

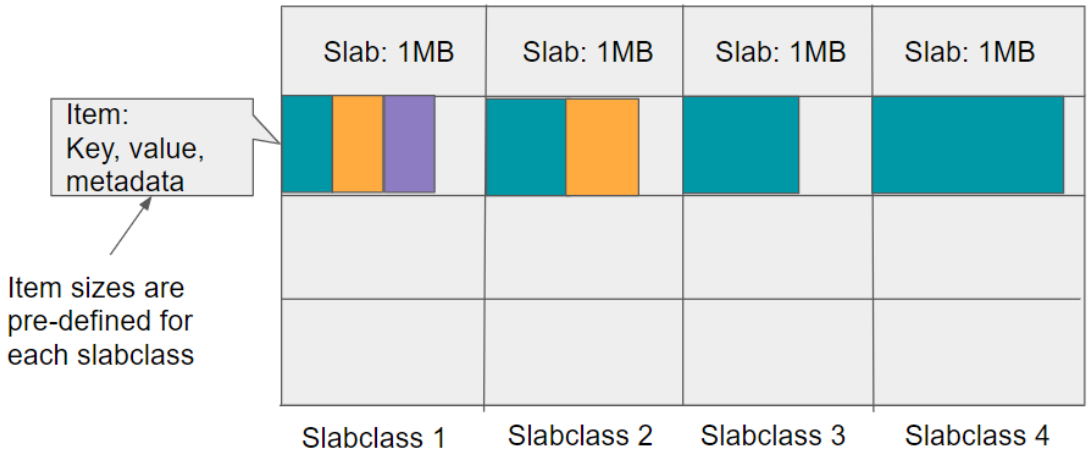


Figure 2.2: A description of Memcached's data structures to store objects.

default, the size of each slab is set to 1MB. Each slab consists of chunks (or items) in which the value of a key is stored along with metadata. Hence, MCD limits the size of an item to be at most the size of a slab. To allow for different value sizes, each slab is assigned with a specific class based on the maximum size of each of its items. Hence, upon an incoming write request, the size of the object to be written is used to identify the best fit item size. Upon a read request, the key is used to find (from a hashtable, described next in section 2.1.1.2) the mapping to the object and, effectively, the slab in which the object is current. Each slab maintains an LRU data structure implemented as a list. Each element in the list is a pointer to an allocated item in the slab. Consequently, an eviction on the LRU list (i.e., from the tail) removes the item from its slab.

2.1.1.2 Flat vs. Segmented LRU in MCD

As mentioned in the previous section, each slab maintains its own LRU list. Each of the LRUs is protected by a lock. Hence, a thread is required to hold the lock prior to accessing a slab. Upon a received request, a worker thread (from a pool) is for assigned to process the request. Referring to figure 2.3, Worker threads use the hashtable to find the slab in which the object resides. Upon finding the slab, the worker thread holds the lock, performs the request and, consequently, updates the state of the LRU. For write requests, the size of the value is used to determine the appropriate slab to be allocated in, along with updating the LRU.

There are two different types of LRU lists in which a MCD server can operate on: Flat-LRU and Segmented-LRU. When operating under Flat-LRU, the worker thread simply evicts the element at the tail of the list when the slab is full to accomedate for the new item. Consequently, The evicted item is then freed from the hashtable as well. On a cache hit, the item is placed at the head of the list. Referring to figure 2.3, as each slab has an LRU list, a hashtable is used to serve read hits (i.e. the hashtable consists of pointers to items). Conflicting items in the hashtable are stored as a linked list. Note that, typically, there are multiple slabs with the same slabclass to allow parallelism within items of the same size.

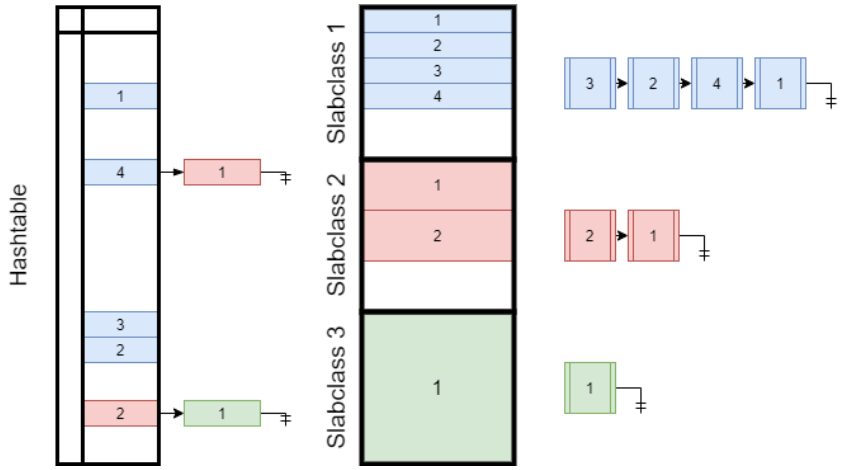


Figure 2.3: An overall diagram that describes the internals of MCD.

The segmented-LRU list consist of four main segments (queues) in which an item can be placed. Figure 2.4 [1] illustrates the transitions between states of an item in a slab which corresponds to its position in the list. In practice, the transitions occur periodically and is maintained by a maintainer thread (explained in Section 2.1.1.3). As illustrated in figure 2.4, when a new item is created with a short Time-to-Live (TTL), it is placed in TEMP queue; TTLs are usually defined in seconds (transition 2). Otherwise, the new item is placed in the HOT queue (transition 1). Items in the HOT queue are never promoted to the head. On the other hand, *active* items that are the tail of the HOT queue are moved to the head of the WARM queue (transition 3); *none active* items are moved to the head of the COLD queue(transition 5). An item is considered to be active if it has been

[1]Figure borrowed from `memcached.org/blog/modern-lru/`

accessed at least twice. Similarly, non-active items that are at the tail of WARM queue are also moved to the COLD queue (transition 7). However, active items that are at the tail of the WARM queue are promoted to the head of the WARM queue (transition 4), the same transiton occurs with active items in the COLD queue (transition 6). An eviction in the COLD queue occurs when the slab is full and a new item is needed to be inserted.

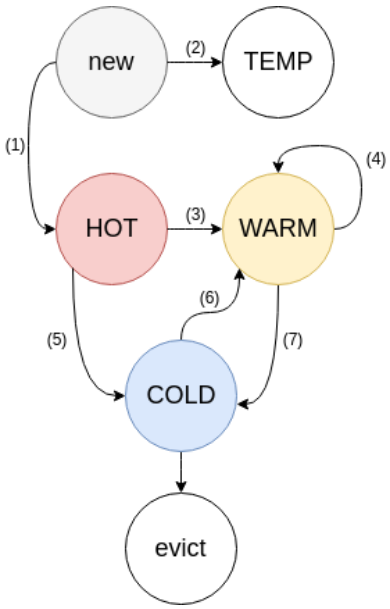


Figure 2.4: A state machine representation of an object in the system from creation to eviction

2.1.1.3 LRU Maintainer

The state transitions of items in the Segmented-LRU setting are maintained by a background thread; referred to as the maintainer. MCD operates under Segmented LRU when the maintainer thread is enabled. The first job of the maintainer is to iterate through every LRU buffer (i.e., the TEMP, HOT, WARM and COLD buffers) and observe the tail. The maintainer performs the transition of tailed element as mentioned in 2.1.1.2. Second, expired items are also removed by the maintainer (expired items were set with a time-to-live with a SET command). Lastly, the maintainer moves items across buffers to respect limits such as space allocation for each of the queues.

2.1.2 Caching with Object Sharing

A naive way to allow for object sharing is to allow requests from all proxy servers to be processed by a single cache (i.e., with a single LRU). However, such naive design does not provide guarantees on isolation between the sharing proxies (i.e., no proxy server should recive less cache space than what it is paid for) [4,6]. There are many prior work such as [11–13] have introduced object sharing. In [6], for example, they consider a single LRU list with head pointers for each proxy are totally ordered; such that higher priority proxies are at earlier elementes of the list. In this work, we consider the object sharing framework as described in [8]. Consider a system with the set J proxies that services database requests from users. Also, consider a database with the size of N data objects and a cache of size B data objects (where $B \ll N$). For each proxy i in the system, an allocation of b_i is set such that $\sum_{i=1}^{|J|} b_i \leq B$. Let $\mathcal{P}(n) \subset J$ be the set of proxies for which object n currently appears in their LRU-list, where $\mathcal{P}(n) = \emptyset$ if and only if object n is not *physically* cached. In other words, the object is not in any of the LRU-lists in the system. Let l_n be the size of the object n. Upon a request by a proxy i for object n, it is inserted at the head of the LRU-list for proxy i. If the request was a hit, then no further actions is needed. Otherwise, the object is added to the set $\mathcal{P}(n)$. Furthermore, we subtract $l_n/|\mathcal{P}(n)|$ from the available cache size of proxy i. as mentioned in [4], we prevent clients from "playing" the system into their favor by performing the following on a read miss:

- if the object is not stored in the physical cache then it is fetched from the database, stored in the cache and forwarded to proxy i;
- otherwise, the object is produced for proxy i after an equivalent delay.

2.2 System Design and Implementation

We implement MCD-OS by modifying the original code of MCD Version 1.5.16. All changes were made on the server side code. We use the same data structure that MCD uses to store the objects (or items). We use the inherent hashtable data structure in MCD mapping keys to items. In order to distinguish between clients, requests from a single client is handled solely by a dedicated worker thread.

Consequently, the thread is responsible for updating the state of all LRUs, along with metadata about proxy servers allocations. The number of threads in the system is predefined with the amount of cache allocated for each proxy server. Each of the LRU lists on the left of Figure 2.5 depicts the LRU state for a proxy (e.g., LRU list 1 is attributed to proxy server 1).

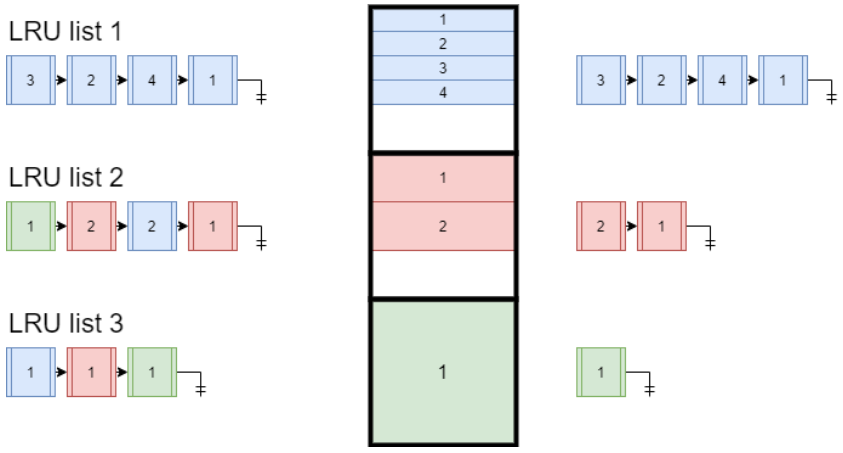


Figure 2.5: MemCacheD with Object Sharing (MCD-OS)

2.2.1 LRUs with Object Sharing

We replace the doubly linked list of pointers in the item data structure to an array of double linked list. The indices to the array is equivalent to proxies ids. If the item does not exist in proxy i's LRU, then the i^{th} is set to *NULL*.

2.2.1.1 Flat-LRU

When processing a read request from client i under the Flat-LRU, under a LRU hit, the LRU list for client i is updated by simply promoting the item to the head. Such updates does not affect the state of the other LRUs in the system (for every LRU j where $i \neq j$); similar to evicted items that are not shared across different clients. We use the term *ripple effect* to describe the phenomena when an eviction in one LRU list causes an eviction at others. *ripple effect length* is the number of LRUs that have been touched when an eviction occurs (more details presented later in section 2.3).

2.2.1.2 Segmented-LRU

Similar to MCD, we implement Segmented-LRU with object sharing. The Segmented-LRU in MCD-OS consist of the four main queues: *HOT*, *WARM*, *COLD* and *TEMPORARY* (*TEMP*). Note that, unlike in MCD, a Segmented-LRU is allocated for each proxy in the system. For a single LRU, the size of each queue is set to be proportional to its allocated proxy cache size. By default, the size of each of the `HOT` and `WARM` queues are set to 32% of the memory allocated to the proxy. The rest is used to allocate the `COLD` queue. For implementation simplicity, the size of the TEMP queue is set to be unlimited for each proxy since they have very short TTL, usually defined by the application. Changing the state of items are handled by the LRU maintainer as described in section 2.1.1.3. Note that with object sharing in play, an item can inflate in the virtual size and violate the size of a segment. Such violation is eventually corrected by the LRU maintainer.

2.2.2 Server Modification on GETs/SETs

The entire set of LRUs is protected by a global lock. Upon receiving a request, a worker thread (dedicated to the proxy sending the request) holds the lock and updates the set of LRUs accordingly (mainly, LRUs in which the object is simultaneously appearing). It is important to note that, unlike MCD, the lock is global and only one thread at a time can access the set of LRU-lists since updates on LRUs are required to be done atomically. In order to implement object sharing using MCD's native API, we modify the two main methods that update LRUs states. Algorithm 1 describes our modification on the MCD native method *evict*. This method is invoked to evict an item from a specific LRU. An important thing to note is that when there are no longer proxies that share the object, the item is freed from the physical cache and the hashtable. Otherwise, Algorithm 2 is invoked which increases the virtual size of the object.

Algorithm 1 evict (item *it, libevent_thread *k)

//unlink item it from proxy-LRU k

$k \rightarrow$ attrib_size -= $it \rightarrow$ virtual_len

$k \rightarrow$ avail_size += $it \rightarrow$ virtual_len

$\mathcal{P}$ = {proxies sharing item it}

if $\mathcal{P} == \emptyset$ **then**
  remove item it from hashtable; free item it in physical cache
else
  inflate (it, $\mathcal{P}$)
end

Algorithm 2 inflate (item *it, libevent_thread *k)

$N = |\mathcal{P}|$ `// P contains all proxies sharing item it`

diff = $\frac{it \rightarrow actual\_len}{N}$ - $it \rightarrow$ virtual_len $it \rightarrow$ virtual_len = $\frac{it \rightarrow actual\_len}{N}$

for i *in* $\mathcal{P}$ **do**
  i $\rightarrow$ attrib_size += diff i $\rightarrow$ avail_size -= diff
end
for i *in* $\mathcal{P}$ **do**
  `// check whether proxy i has enough space after inflation, may`
  `// evict more than one item in proxy-LRU i`
  while $i \rightarrow$ *attrib_size* $>$ $i \rightarrow$ *alloc_size* **do**
    evict(tails[i], i) `// evict the tail item of proxy-LRU i if`
    `// it's full`
  end
end

Algorithm 3 is another MCD native method that we have modified to enable object sharing. For LRU k, the method is invoked to either insert a new item or promote an existing item to the head of the LRU. Updating the state of other LRUs (that are not k) only occur when the virtual size of an item has changed. Otherwise, only one LRU is updated. Algorithms 2 and 4 are methods that enforce allocation limits across proxies. We can observe that algorithm 1 is also invoked in algorithm 2 causing the ripple effect (further discussion in section 2.3).

Algorithm 3 insert (item *it, libevent_thread *k, bool new)

```
// insert item it to head of proxy-LRU k; insert item it in
// hashtable
/* update statuses of proxies sharing item it                    */
```
$\mathcal{P}$ = {proxies sharing item it}
    N = $|\mathcal{P}|$ `// at least 1 since item it is inserted to k`
old_virtual_len = it→virtual_len
 new_virtual_len = $\frac{it \rightarrow actual\_len}{N}$
 diff = old_virtual_len - new_virtual_len
 if *new* **then**
    `// item it is new to proxy-LRU k`
    k→attrib_size += old_virtual_len
     k→avail_size -= old_virtual_len
end
switch *diff* **do**
    case *diff == 0* **do**
        `// virtual length doesn't change, check whether insertion`
        `// causes eviction in LRU k, no effect on other LRUs`
        while $k \rightarrow attrib\_size > k \rightarrow alloc\_size$ **do**
            evict(tails[k], k) `// evict the tail item if LRU k is full`
        end
    end
    case *diff > 0* **do**
        deflate(it, k, $\mathcal{P}$) `// decreased virtual length causes deflation`
    end
    case *diff < 0* **do**
        inflate(it, $\mathcal{P}$) `// increased virtual length causes inflation`
    end
end

Algorithm 4 deflate (item *it, libevent_thread *k, libevent_thread ** $\mathcal{P}$)

$N = |\mathcal{P}|$ `// ` $\mathcal{P}$ `contains all proxies sharing item it`

diff = $it \rightarrow$ virtual_len - $\frac{it \rightarrow actual\_len}{N}$ $it \rightarrow$ virtual_len = $\frac{it \rightarrow actual\_len}{N}$ **for** i *in* $\mathcal{P}$ **do**

| i $\rightarrow$ attrib_size -= diff i $\rightarrow$ avail_size += diff

end

while $k \rightarrow$ *attrib_size* $> k \rightarrow$ *alloc_size* **do**

| `// check whether proxy k has enough space for the new item, may evict more than one object in proxy-LRU k`

| evict(tails[k], k) `// evict the tail item of proxy-LRU k if it's full`

end

Table 2.1 summarizes the different behavior in response to SET/GET requests from a proxy. Here, we consider multiple scenarios in terms of the presence of an object (wither the object is present in the physical cache, LRU list or both.) and the command received.

2.2.3 Handling Objects with Different Sizes

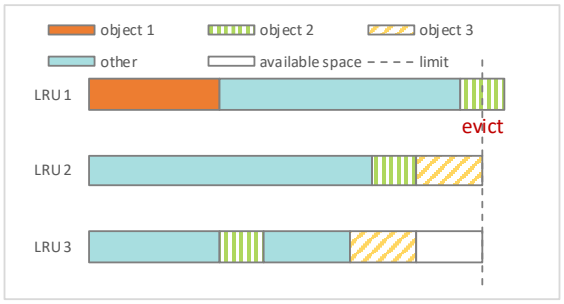
As described in section 2.2.1, there are two different sets of LRU lists when running an MCD-OS instance. The first set of LRUs map to proxy servers (hence, referred to as proxy LRUs). These LRUs are provisioned based on the amount of cache allocated per proxy. Different objects with different physical lengths can co-exist in the same proxy LRU. On the other hand, there exist another set of LRU lists that are mapped to slabs (referred to as slab LRUs); only objects with the same physical length can co-exist in the same slab LRU. Slab LRUs is used to implement an eviction policy that is based on slab accesses. Promoting items in the slab LRUs does not affect the state of proxy LRUs. However, when an item is evicted from a slab LRU, then the object is removed of all proxy LRUs that share the object (remitting the virtual size to every proxy LRU sharing the object). Such evictions only occur when a new item is needed to be allocated with no existing available space on a slab with the same class.

proxy i issues `get(k)`; hits in LRU i
• promote item with key k to the head of LRU i
proxy i issues `get(k)`; misses in LRU i but hits in cache
• insert the item with key k into the head of LRU i • update the status of all other LRUs sharing this item (deflation)
proxy i issues `get(k)`; misses in both LRU i and cache
• return cache miss to client // client is expected to fetch the item from database and issue `set(k, v)`
proxy i issues `set(k, v)`; key k doesn't exist in cache
• package the key-value pair `(k,v)` into an item, store in cache • set virtual length of the item to its actual length • insert the item to head of LRU i
proxy i issues `set(k, v)`; key k already exists in cache
• update the item with key k to reflect the new value `v` • promote the item to head of LRU i • update the status of all other LRUs sharing this item (may involve a combination of inflation and deflation)

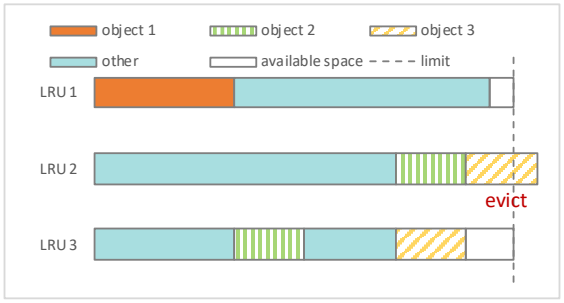
Table 2.1: A Summary of MCD-OS response behavior.

2.3 Overheads of Object Sharing in MCD-OS

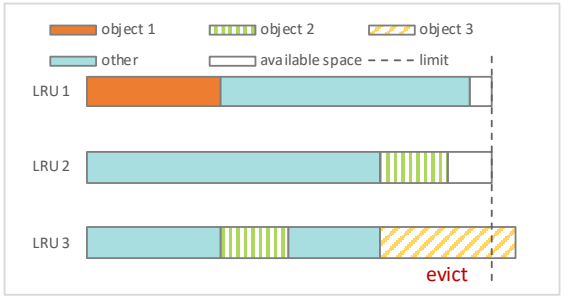
This section describes the overheads of MCD-OS. Specifically, the change of virtual sizes of objects can violate LRU allocations resulting in a series of evictions, which we refer to as the ripple effect. With object sharing, the virtual size of an object is affected by the number of proxy servers that are effectively sharing the object. *Ripple length* is the number of eviction occurred due to a ripple effect. *Inflation* is an increase in the virtual size of an object due to an eviction (i.e., the decrease in the degree of sharing). Figure 2.6 describes an example of an object inflating causing a ripple effect with length of 3. On the other hand, *deflation* (figure 2.7) is the decrease in the virtual size of an object due the increase in the degree of sharing.



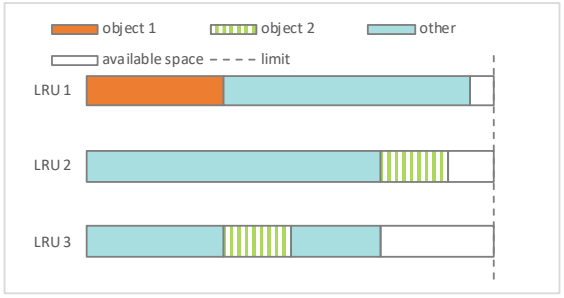
(a) Assume we have three equal sized LRUs. Object 2 is shared by LRU 1, 2, and 3. Object 3 is shared by LRU 2 and 3. A new item, object 1, is inserted to the head of LRU 1.



(b) LRU 1 evicts object 2. So, the virtual length of object 2 inflates in LRUs 2 and 3. So, LRU 2 exceeds its limit and needs to evict object 3.

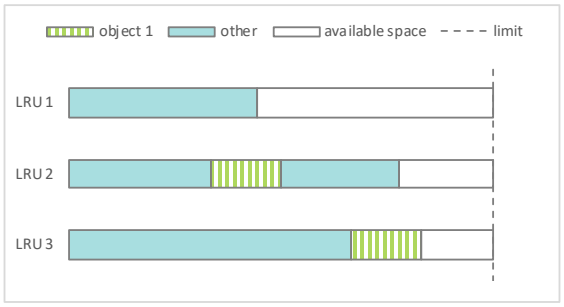


(c) The increased virtual length of object 3 similarly requires LRU 3 to evict.



(d) LRU 3 evicts object 3. Now no LRU exceeds its limit and processing of the insertion of object 1 into LRU 1 in (a) is completed.

Figure 2.6: Series of proxies LRUs states describing an object inflating causing a ripple effect.



(a) Assume we have three equal sized LRUs. Object 1 is shared among LRU 2 and 3 only.

(b) LRU 1 inserts object 1 into it LRU. Hence, the virtual length of object 1 deflates.

Figure 2.7: Searies of proxies LRUs states describing an object deflating.

2.4 Evaluation

We use our prototype of MCD-OS to evaluate the hit rates and the ripple effect when enabling object sharing. The workload is generated by specifying the Zipfian parameter α to describe the key popularity skew. We model each proxy as a thread/process that sends it requests to MCD-OS based on its assigned Zipfian α parameter. In terms of server side of MCD-OS, each proxy is mapped to a single worker thread that is responsible to process the request and maintain the states of the LRU-lists. In this section, we first validate our prototype (section 2.4.1) by comparing our results to [8]. We validate our prototype under both LRU settings (Flat and Segmented). Second, we evaluate the extensiveness of the MCD-OS overheads in section 2.4.2.

cli	alpha	mem	h_1	h_10	h_100	h_1000
0	0.75	8	0.3651	0.0740	0.0129	0.0021
1	0.5	8	0.1256	0.0376	0.0130	0.0040
2	1	8	0.6491	0.1090	0.0116	0.0011
0	0.75	8	0.3819	0.0814	0.0145	0.0026
1	0.5	8	0.1276	0.0424	0.0134	0.0042
2	1	64	0.9267	0.7172	0.1270	0.0137
0	0.75	8	0.3781	0.0749	0.0139	0.0027
1	0.5	64	0.6654	0.2872	0.0964	0.0335
2	1	8	0.6720	0.1181	0.0120	0.0014
0	0.75	8	0.4040	0.0897	0.0163	0.0027
1	0.5	64	0.6923	0.2985	0.1120	0.0342
2	1	64	0.9331	0.7336	0.1321	0.0147
0	0.75	64	0.9728	0.4909	0.1102	0.0218
1	0.5	8	0.1249	0.0427	0.0126	0.0042
2	1	8	0.7246	0.1357	0.0142	0.0016
0	0.75	64	0.9839	0.5356	0.1312	0.0240
1	0.5	8	0.1302	0.0474	0.0136	0.0043
2	1	64	0.9543	0.7914	0.1317	0.0133
0	0.75	64	0.9025	0.5103	0.1173	0.0220
1	0.5	64	0.6802	0.2987	0.1074	0.0357
2	1	8	0.7241	0.1397	0.0143	0.0014
0	0.75	64	0.9721	0.5381	0.1413	0.0259
1	0.5	64	0.7044	0.3068	0.1119	0.0374
2	1	64	0.9313	0.8188	0.1504	0.0129

Table 2.2: Flat LRU hit rates under MCD-OS

2.4.1 Prototype Validation

We run our validation experiments on Amazon Web Services (AWS). We run each of the server and clients on two separate VMs on a single region. We construct our workload to emulate a system with three proxies where clients are characterized by a zipfian distribution. We compare our prototype results with [8].

cli	alpha	mem	h_1	h_10	h_100	h_1000
0	0.75	8	0.3799	0.0909	0.0331	0.0157
1	0.5	8	0.1504	0.0611	0.0320	0.0335
2	1	8	0.6706	0.1175	0.0396	0.0171
0	0.75	8	0.4034	0.1039	0.0381	0.0050
1	0.5	8	0.1333	0.0545	0.0330	0.0217
2	1	64	0.9530	0.7443	0.1459	0.0287
0	0.75	8	0.3920	0.0837	0.0210	0.0304
1	0.5	64	0.6823	0.3151	0.1206	0.0532
2	1	8	0.6965	0.1331	0.0280	0.0219
0	0.75	8	0.4200	0.1195	0.0222	0.0098
1	0.5	64	0.7129	0.2997	0.1240	0.0461
2	1	64	0.9566	0.7487	0.1422	0.0269
0	0.75	64	0.9931	0.4994	0.1368	0.0269
1	0.5	8	0.1271	0.0513	0.0157	0.0133
2	1	8	0.7282	0.1417	0.0144	0.0278
0	0.75	64	0.9923	0.5418	0.1581	0.0353
1	0.5	8	0.1385	0.0514	0.0253	0.0056
2	1	64	0.9803	0.8039	0.1504	0.0222
0	0.75	64	0.9249	0.5222	0.1253	0.0334
1	0.5	64	0.6830	0.3199	0.1113	0.0525
2	1	8	0.7441	0.1493	0.0316	0.0301
0	0.75	64	0.9944	0.5590	0.1709	0.0551
1	0.5	64	0.7261	0.3246	0.1176	0.0569
2	1	64	0.9448	0.8447	0.1718	0.0320

Table 2.3: Segmented LRU hit rates under MCD-OS

2.4.2 Ripple Effect

We measure the overhead of a ripple by counting the number of evictions in a ripple. We refer to the number of evictions as the length of the ripple. Note that a ripple effect with length of 1 means that only a single LRU has evicted an item. Another thing to note is that it is possible for a ripple length to exceed the total number of LRU-list in the system caused by an object inflation, as mentioned in section 2.3. We run our experiment to evaluate the ripple effect overhead on AWS EC2 [14]. We launched two VMs (a server and client instances) of type *m5.xlarge* in a single datacenter. On the client side, we used $J = 9$ proxies with $N = 10^6$ items, where each item was 100kB. The total cache memory was 3 GB. For each proxy $i \subset [J]$,

its zipf parameter is set to $0.5 + 0.5(i - 1)$ and memory allocation: b=100 MB for proxies 1,2,3; b=200 MB for proxies 4,5,6; and b=700 MB for proxies 7,8,9. The clients issue 3×10^6 GET commands in total, after the cold misses have abated. On read misses, clients perform a SET command to update the cache with the value. We record the ripple effect caused by an eviction when a SET command is issued 2.8. The horizontal axis represents the number of evictions that occured in the ripple; we refer to it also as the length of the ripple. We can see that only 16% of the SET commands experience a ripple length that is greater than one. In addition, we record the latencies of SET requests and evaluate the overhead of updating the LRU lists due to ripple effects, as shown in figure 2.9.

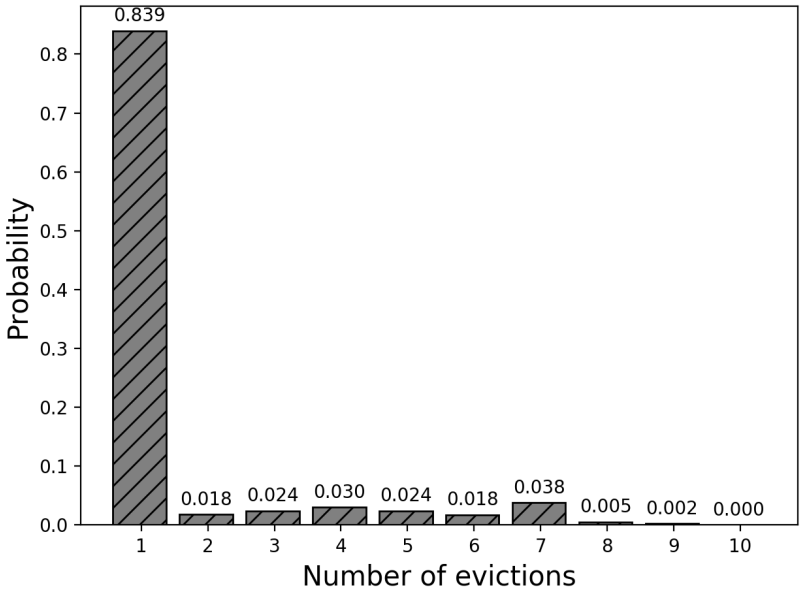


Figure 2.8: Ripple effect evaluation using MCD-OS.

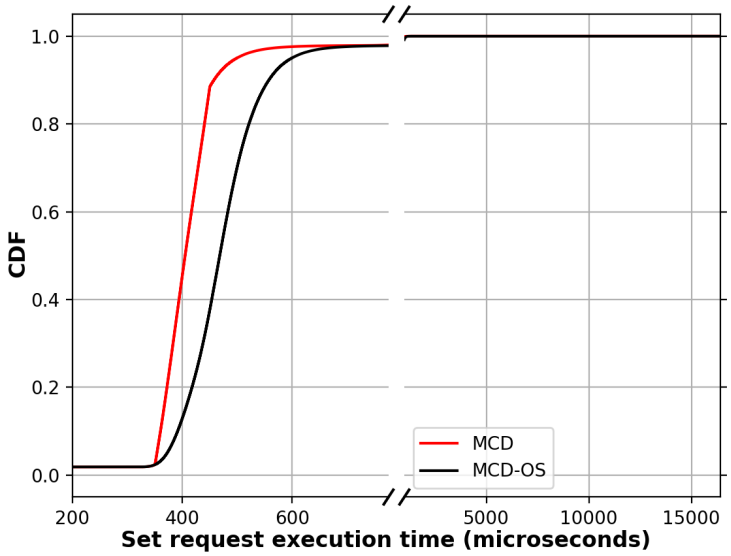


Figure 2.9: CDF of the `set` requests in MCD vs. MCD-OS

Chapter 3
Optimization Across Multiple Regions

In this thesis, we are interested in geographically distributed key-value storage system that guarantees strong consistency (linearizability) [15]. Such systems are naturally geographically distributed in order to tolerate datacenter failures [16, 17]. Hence, accessing other datacenters is necessitated in order to enforce linearizability. While the simplest form of tolerating failure is through replication, erasure coding is yet another form of distributed storage scheme in which a failure can be tolerated. Erasure coding has been proven to be an effective scheme to provide a KV storage system that guarantees linearizability while minimizing storage and communication cost and meeting SLO [5, 7, 18]. For this work, we implement our prototype to include optimistic protocols in order to minimize cost (described in section 3.2.1). The challenge of such optimization is to navigate a complex space of trade-offs due to the inherent diversity in latency and costs that are inevitable for the public cloud, as well as a wide range of parameters of workload characteristics and SLO requirements. For this, we use the optimization formulation mentioned in [18] with a modification on the heuristic in order to minimize cost. Specifically, we use a network price based cost heuristic (i.e., quorum selection decisions are based of data centers that are cheapest in network cost; described in section 3.2.2).

3.1 Background and Related Work

3.1.1 Erasure Coding

Erasure coding is a form of data resiliency in which data is fragmented into k segments. In addition, there are m parity fragments formed from linear combinations of the k data fragments. In *maximum distance separable* (MDS) codes, the data can be recovered as long as there are any k available fragments among the $k + m$. Similar to replication based protocols, erasure coding is used to tolerate failure while preserving strong consistency [19–22]. Clearly, erasure coding promotes storage savings when splitting data into smaller chunks. For the same reason, we can minimize the cost attributed to network, as mentioned in 3.2.2. There is a substantial prior work on using erasure coding for storage efficiency. We find that [23] to be the most relevant to our work.

3.1.2 Linearizability

Linearizabilty is a protocol property in which all operations are seen by all processors in the same order [15]. In this section, we describe the two protocols that we use to implement a strongly consistent geo-distributed storage system. Hence, we describe both of the protocols in the semantics of `GET`/`PUT` requests. For this work, it is important to note the following properties when describing the protocols: number of rounds in which the protocol executes requests and identifying the source/destination of the datagrams. These properties used in our optimization formulation; also mentioned in the optimization of [18]. In addition, note that both protocols explored in this thesis are non-blocking. In other words, operations terminate in fixed number of rounds (2~3 rounds) even in the face of concurrent reads or writes. We associate each version of a key with a *tag* (also refer to as *timestamp*). The *timestamp* is an element of a *totally ordered* set. In practice, the timestamp consist of two parts: an integer component (referred to as the logical time) and client id.

3.1.2.1 ABD Protocol

ABD[1] is a replication and quorum based protocol that ensures strong consistency while tolerating crash failures of nodes. A quorum in ABD is simply defined as a majority of the servers. In our work, however, a quorum is going to be selected based on the network cost associated to the phase (described in section 3.2.2).

PUT(*key*, *value*):

get-timestamp: Send *get-timestamp* message to all server and wait for a majority to respond with *timestamp*. After receiving responses, select the largest timestamp and let its integer component be t.

put-value: Create a new timestamp *timestamp-new* as ($t + 1$, *client-id*) where *client-id* is an unique id associated with each client. Send *put-value* message with (*timestamp-new*, *value*) to all servers and wait for a majority to respond. Upon receiving all responses from the write quorum, return to client.

Figure 3.1: Client Side PUT Protocol for ABD

GET(*key*):

get-timestamp-value: Send *get-timestamp-value* message to all servers and wait for a majority to respond with (*tag*, *value*). After receiving responses, select the largest timestamp and its corresponding value (*max-tag*, *max-value*).

put-value: Send *put-value* message with (*max-tag*, *max-value*) to all servers and wait for a majority to respond. Return *max-value* to client.

Figure 3.2: Client Side GET Protocol for ABD

[1] Acronym driven from the authors' name of [24]. We use the term to refer to a multi-writer variant of the algorithm of [24], which can be found in [25].

Server Side Protocol:
Initial State: Stores the (*timestamp*, *value*) pair for each key.

On receiving get-timestamp: Respond back with highest locally available *timestamp*.

On receiving get-timestamp-value: Respond back with highest locally available *timestamp* and its *value*.

On receiving put-value: If received timestamp is greater than locally available timestamp, store new (*timestamp*, *value*). Respond back with acknowledgement in any case.

Figure 3.3: Server Side Protocol for ABD

3.1.2.2 CAS Protocol

CAS[2] [23, 26] is a quorum based protocol that utilizes erasure coding to minimize storage cost while guaranteeing SLO. We describe the protocols using the following notation. We denote the encoding parameter as (n, k), where n is the number and k is the number of coded elements needed to retrieve the data. In addition, we use Q_i for $1 \leq i \leq 4$ to indicate the quorum size needed to receive response from. A configuration with (n, k) is linearizable if the following constraints are met:

- $Q_1 + Q_3 > n$
- $Q_2 + Q_4 \geq n + k$
- $Q_1 + Q_4 > n$
- $Q_4 \geq k$

Lastly, when using the CAS protocol, we include an additional state bit S to every data chunk such that $S \in$ {'pre', 'fin'}

[2]The acronym stands for CAS *Coded Atomic Storage*

PUT(*key*, *value*):

get-timestamp: Send *get-timestamp* message to all servers and wait for Q_1 of them to respond with *timestamp*. After receiving responses, select the largest timestamp. Let t be the integer component of the highest timestamp recived. Apply the MDS coding (from section 3.1.1) using (n, k) on the *value*; creating data chunk of $C_1, ..., C_n$.

put-code: Create a new timestamp *timestamp-new* as ($t + 1$, *client-id*). Send *put-code* message with (*timestamp-new*, C_i, 'pre') to i^{th} server) for $1 \leq i \leq n$. Await responses from Q_2.

put-fin: Send *put-fin* message all servers with *timestamp-new*. Await for Q_3 to respond.

Figure 3.4: Client Side PUT Protocol for CAS

GET(*key*):

get-timestamp: Send *get-timestamp* message to the Q_1 quorum servers and wait for all of them to respond with *timestamp*. On receiving response, select the largest timestamp, lets call it *timestamp-max*.

get-code: Send *get-code* message with *timestamp-max* to the Q_4 quorum and wait for all of them to respond with their corresponding *codes*. Decode the *value* from collected *codes*.

Figure 3.5: Client Side GET Protocol for CAS

Server Side Protocol for CAS:
Initial State: For each key, it stores (*timestamp*, V, S) where $V \in \{C_i, \text{'null'}\}$, C_i is the coded element for server i. $S \in \{\text{'fin'}, \text{'pre'}\}$.

On receiving get-timestamp: Respond with locally known most recent *timestamp* for the key with the 'fin' tag.

On receiving get-code: If the (*timestamp*, C_i, *) exits for the key where * could be 'pre' or 'fin' then send this code else add (*timestamp*,'null' , 'fin') and send 'pre'.

On receiving put-code: If the *timestamp* doesn't exits for the key then update (*timestamp*, C_i, 'pre') else ignore. Send acknowledgement in any case.

On receiving put-fin: If (*timestamp*, C_i, 'pre') exists for the key then update it to (*timestamp*, C_i, 'fin') else insert (*timestamp*, 'null', 'fin'). Send acknowledgement in any case.

Figure 3.6: Server Side Protocol for CAS

3.2 System Design

3.2.1 Optimistic Reads

As described in section 3.1.2.1, ABD reads consist of two phases. In both phases, the value is sent via the network incurring twice the cost. Specifically, on the first phase, the client receives a set of responses that consist of the timestamps and their values from the read quorum. On the second phase, the client sends the value with the highest timestamp to the write quorum. In this work, we optimize ABD reads to return after the first phase when all values received , from the read quorum, are for the same version (i.e., they all have the same timestamp). We argue that our optimization does not violate strong consistency due to the read/write quorum construction, where Q_{read} and Q_{write} are the sets for each quorum and $|Q_{read} \cap Q_{write}| \geq 1$. Figure 3.7 describes the client side for ABD's GET request optimization.

OPT-GET(*key*):

get-timestamp-value: Send *get-timestamp-value* message to the read quorum and wait for all of them to respond with (*tag*, *value*). On receiving response, if all the responses are for the same version, then return to client the *value*. Otherwise, select the largest timestamp and its corresponding value (*max-tag*, *max-value*).

put-value: Send *put-value* message with (*max-tag*, *max-value*) to the write quorum and wait for all of them to respond. Return *max-value* to client.

Figure 3.7: Client side optimistic GET requests for ABD

Similarly, CAS reads consist of two phases (as describe in section 3.1.2.2). The second phase of CAS is optimized such we optimistically only contact k servers that are the cheapest in terms of network. However, unlike ABD, we implement CAS such that terminates in two phases in any case.

3.2.2 Network Cost Heuristic Optimization

We use the formulation mentioned in [18] and modify our heuristic such that it would select data centers that are cheaper in terms of their bandwidth cost. We argue that such heuristic can be more cost-effective to utilized in the public cloud; while certainly meeting SLOs. Figure 3.8 and 3.9 illustrates two scenarios in which the nearest data center heuristic mentioned in [18] is costlier than the configuration given by our optimizer. Figure 3.8 illustrates a scenario where both configurations uses ABD. However, the data was placed in different data centers across configuration. In this scenario, we consider a workload with a skewed popularity of clients (90 percent of the request are generated from japan) and 90 percent of the requests are reads. Similarly, Figure 3.9 considers a workload where 30 percent of the requests are originating from Japan, and 60 percent are originating from São Paulo. Using the configuration provided by the optimizer, we note that despite of having a larger set of data centers that participate in storing the data, erasure coding effectively minimizes the cost of network and storage cost.

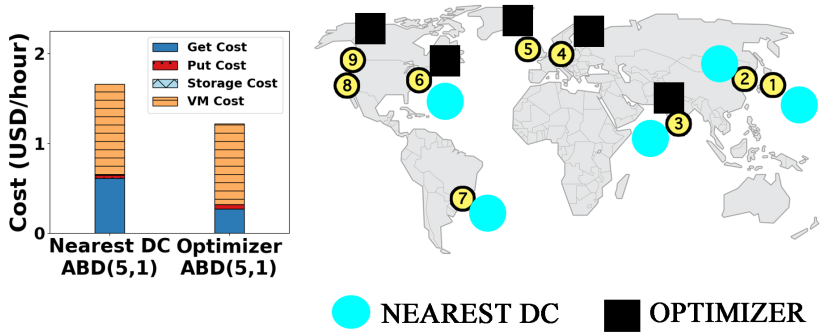


Figure 3.8: Selecting nearest DCs using ABD

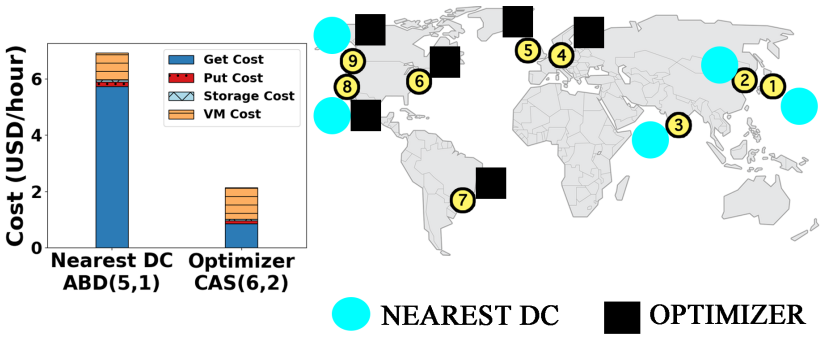


Figure 3.9: Selecting nearest DCs with CAS

3.2.3 Workload Characterization

Similar to the optimization formulation mentioned in [18], we consider dividing the keys into multiple groups. Keys that are in the same group theoretically observe the same behavior. Such simplification allows us to optimize a group of keys separately (optimization mentioned in section 3.2.2) instead of per key-based. Each group is defined by the number of keys, object size, arrival rate of requests, client distribution and read to write ratio.

3.3 System Overheads

3.3.1 Garbage Collection

In CAS, the server does not replace its codeword with an older version, instead it stores a list of all codewords which the server has received. Such action is crucial to refrain from failures due to inconsistent codewords received by the client. A crucial component of this work is to develop a garbage collector (GC) that deletes older versions to keep the storage cost in check. Theory shows a close connection between the storage cost incurred and the amount of concurrency that a system allows; specifically, an algorithm that allows ν concurrent write operations requires to store, in the *worst-case*, $\nu + 1$ versions at each server (See CASGC of [23, 26]). If the concurrency exceeds ν, then a read operation that is concurrent with these writes may not terminate. An important task is to examine the impact of the results of [23, 27, 28] in our implementation. An aggressive GC strategy that keeps very few older versions at the servers would reduce storage cost; however, the result of [27, 28] implies that it can affect performance by preventing the termination of operations. While it might be tempting to tune the number of versions to the extent of concurrency, there are two problems with this approach. First, the worst-case concurrency of a group is difficult to characterize, as it is an implied quantity based on the arrival process of the reads and writes. Second, even if the concurrency v is measured perfectly, the strategy of storing $v + 1$ number of versions can be too conservative and leave money on the table; this is because operation termination is theoretically affected only under a worst-case scenario, and its impact in a practical system is not clear. Finally, we also aim to build a garbage collector that does not consume too many system resources and affect performance. We propose the following heuristics to garbage collect older versions:

- *Number of versions based:* only keeps up to an operator-specified threshold on number of versions of an object.
- *Periodic:* simply scans old versions periodically (once every 5 minutes in our prototype) and removes them.

3.3.2 Reconfiguration

Significant changes in workload properties may render the current configuration expensive or cause SLO violations or both. We exploit predictable patterns in workloads to identify when such changes are likely to occur and invokes its optimizer with revised estimates of workload properties (over the foreseeable near-term period of relative workload stationarity), thereby starting a new epoch. In addition to such predictive mechanisms, a new epoch may also be started by in a "reactive" manner upon detecting unsatisfactory system behavior (SLO violations or higher-than predicted costs); similar combinations of predictive and reactive control comprise a well-studied area [29, 30].

3.4 Evaluation

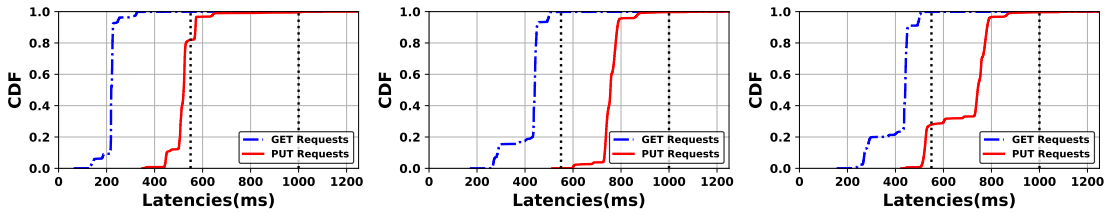


Figure 3.10: GET/PUT latency distributions observed for baseline 1 (left), baseline 2 (middle), and our prototype (right).

	Group		
	g=1	g=2	g=3
Baseline 1	ABD(5,1)	ABD(5,1)	ABD(5,1)
Baseline 2	CAS(6,2)	CAS(5,1)	CAS(5,1)
Our Prototype	CAS(6,2)	CAS(5,1)	ABD(5,1)

Table 3.1: Group characterization on workload evaluated.

We conduct our experiments on AWS using data collected from [31, 32]. We present 3 sample results, each comparing the estimates for the following vs. observations on our prototype: baseline 1, baseline 2, and our prototype. The workload comprised three groups (table 3.1), one of which was write-heavy and the remaining read-heavy. The workloads had skewed spatial distributions of users and contains object sizes in the range of 1-50KB. Baseline 1 is constructed such that our optimization

	Cost (USD/hour)				Performance (ms)			
	VM		Network		GET		PUT	
	Est.	Obs.	Est.	Obs.	Est.	Obs.	Est.	Obs.
Baseline 1	1.32	1.32	1.54	1.24	548	322	548	572
Baseline 2	1.32	1.32	0.85	0.94	548	500	792	868
Our Prototype	1.32	1.32	0.83	0.72	548	502	792	864

Table 3.2: Cost and performance for baseline 1, baseline 2 and our prototype

framework only picks ABD as the protocol along with the best placement policy for each group. Similarly, Baseline 2 only looks at possible configurations using CAS protocol. Lastly, using our prototype, we optimize the placement of each group to select any of the protocols. Table 3.2 depicts the cost breakdown for each of the experiments and its comparison with (approximate) costs incurred as reported by AWS billing and cost management dashboard. We find that both cost and SLO are well-estimated for baselines as well as our prototype. The observed cost and tail latency for baseline 1 are noticeably better than the optimizer's estimate due to cases when the ABD protocol [33] performs its read operation in one phase - clearly our modeling of this phenomenon is on the conservative side. The table also compares a high (98^{th}) percentile of observed request latency and that predicted by the optimizer. Figure 3.10 illustrates the GET/PUT latency distributions observed with configurations chosen by the optimizer. In each case a high percentile of latency is in agreement with the SLO targets specified to the optimizer.

Chapter 4
Conclusion

The attractiveness of migrating all variants of applications to the public cloud have promoted providers to create extensive number of services; allowing tenants to tailor their system design to their needs. In this work, we have focused on minimizing the cost of key-value storage systems in two different scales: single region and across regions. In the former, a key-value system typically consists of a caching unit along with the database. Here, we consider a caching system where different remote proxies accessing the database "pay" for their allocation. We implemented a caching system that enables object sharing to minimize storage cost while respecting their SLAs. We evaluated our system by comparing the performance of the prototype against results provided theory. In addition, we evaluate the extensiveness of the overheads by comparing the results from our prototype to a system without object sharing. Second, we considered KV store systems that are naturally geographically distributed due to the need for tolerating failures. Here, we considered two protocols that guarantees strong consistency: ABD and CAS. The latter employs erasure coding techniques to minimize storage and communication cost. We modified both protocols to perform in an optimistic approach on failure tolerance, while preserving strong consistency. In addition, we considered an optimization formulation that uses network cost as the heuristic in selecting quorums.

Bibliography

[1] Tak, B. C., B. Urgaonkar, and A. Sivasubramaniam (2011) "To move or not to move: The economics of cloud computing," in *Proceedings of the 3rd USENIX conference on Hot topics in cloud computing*, USENIX Association, pp. 5–5.

[2] "Amazon S3," https://aws.amazon.com/s3/.

[3] "Amazon ElastiCache," https://aws.amazon.com/elasticache/.

[4] Pu, Q., H. Li, M. Zaharia, A. Ghodsi, and I. Stoica (2016) "FairRide: Near-Optimal, Fair Cache Sharing," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, USENIX Association, Santa Clara, CA, pp. 393–406.
URL `usenix.org/conference/nsdi16/technical-sessions/presentation/`

[5] Wu, Z., M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha (2013) "SPANStore: Cost-effective Geo-replicated Storage Spanning Multiple Cloud Services," in *Proc. ACM SOSP*.

[6] Quan, G., J. Tan, A. Eryilmaz, and N. Shroff (2019) "A New Flexible Multi-Flow LRU Cache Management Paradigm for Minimizing Misses," *Proc. ACM Meas. Anal. Comput. Syst.*, **3**(2).
URL `https://doi.org/10.1145/3341617.3326154`

[7] Rashmi, K. V., M. Chowdhury, J. Kosaian, I. Stoica, and K. Ramchandran (2016) "EC-Cache: Load-Balanced, Low-Latency Cluster Caching with Online Erasure Coding," in *Proc. USENIX OSDI*.

[8] Kesidis, G., B. Urgaonkar, M. T. Kandemir, and T. Konstantopoulos (2019) "A caching system with object sharing," *CoRR*, **abs/1905.07641**, `1905.07641`.
URL `http://arxiv.org/abs/1905.07641`

[9] "Memcached," https://memcached.org.

[10] Chidambaram, V. and D. Ramamurthi, "Performance Analysis of Memcached," .

[11] Golrezaei, N., K. Shanmugam, A. G. Dimakis, A. F. Molisch, and G. Caire (2012) "FemtoCaching: Wireless video content delivery through distributed caching helpers," in *2012 Proceedings IEEE INFOCOM*, pp. 1107–1115.

[12] Poularakis, K., G. Iosifidis, A. Argyriou, I. Koutsopoulos, and L. Tassiulas (2019) "Distributed Caching Algorithms in the Realm of Layered Video Streaming," *IEEE Transactions on Mobile Computing*, **18**(4), pp. 757–770.

[13] Wang, Y., X. Zhou, M. Sun, L. Zhang, and X. Wu (2016) "A New QoE-Driven Video Cache Management Scheme with Wireless Cloud Computing in Cellular Networks," *Mobile Networks and Applications*, **22**.

[14] "Amazon EC2," https://aws.amazon.com/ec2/.

[15] Herlihy, M. P. and J. M. Wing (1990) "Linearizability: A Correctness Condition for Concurrent Objects," *ACM Trans. Program. Lang. Syst.*, **12**(3), pp. 463–492.

[16] Attiya, H. and J. Welch (2004) *Distributed computing: fundamentals, simulations, and advanced topics*, vol. 19, John Wiley & Sons.

[17] Attiya, H. and J. L. Welch (1994) "Sequential consistency versus linearizability," *ACM Transactions on Computer Systems (TOCS)*, **12**(2), pp. 91–122.

[18] Sharma, C. (2018) "Design And Implementation Of A Cost-Effective Linearizable Geo-Distributed Key-Value Store Combining Replication And Erasure Coding," Master's thesis, available at https://etda.libraries.psu.edu/catalog/15616cks5338.

[19] Cassuto, Y. (2013) "What Can Coding Theory Do for Storage Systems?" *SIGACT News*, **44**(1), pp. 80–88.
URL `http://doi.acm.org/10.1145/2447712.2447734`

[20] Datta, A. and F. Oggier (2013) "An Overview of Codes Tailor-made for Better Repairability in Networked Distributed Storage Systems," *SIGACT News*, **44**(1), pp. 89–105.
URL `http://doi.acm.org/10.1145/2447712.2447735`

[21] Lin, S. and D. J. Costello (2004) *Error Control Coding, Second Edition*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[22] Roth, R. (2006) *Introduction to Coding Theory*, Cambridge University Press, New York, NY, USA.

[23] Cadambe, V. R., N. Lynch, M. Medard, and P. Musial (2014) "A Coded Shared Atomic Memory Algorithm for Message Passing Architectures," in *2014 IEEE 13th International Symposium on Network Computing and Applications (NCA)*, IEEE, pp. 253–260.

[24] Attiya, H., A. Bar-Noy, and D. Dolev (1995) "Sharing Memory Robustly in Message-passing Systems," *J. ACM*, **42**(1), pp. 124–142.

[25] Lynch, N. A. (1996) *Distributed Algorithms*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[26] Cadambe, V. R., N. Lynch, M. Mèdard, and P. Musial (2017) "A Coded Shared Atomic Memory Algorithm for Message Passing Architectures," *Distrib. Comput.*, **30**(1), p. 49–73.
URL `https://doi.org/10.1007/s00446-016-0275-x`

[27] Cadambe, V. R., Z. Wang, and N. Lynch (2016) "Information-Theoretic Lower Bounds on the Storage Cost of Shared Memory Emulation," in *Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, PODC '16, ACM, pp. 305–314.

[28] Spiegelman, A., Y. Cassuto, G. Chockler, and I. Keidar (2016) "Space bounds for reliable storage: Fundamental limits of coding," in *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, ACM, pp. 249–258.

[29] Urgaonkar, B., P. J. Shenoy, A. Chandra, and P. Goyal (2005) "Dynamic Provisioning of Multi-tier Internet Applications," in *Proc. ICAC*.

[30] Gandhi, A., M. Harchol-Balter, R. Raghunathan, and M. A. Kozuch (2012) "AutoScale: Dynamic, Robust Capacity Management for Multi-Tier Data Centers," *ACM Trans. Comput. Syst.*, **30**(4), pp. 14:1–14:26.
URL `https://doi.org/10.1145/2382553.2382556`

[31] Leonhard, M. (2017), "CloudPing. info," .

[32] (last accessed, Sept. 2017), "Amazon EC2 Pricing," `http://aws.amazon.com/ec2/pricing`.

[33] Attiya, H., A. Bar-Noy, and D. Dolev (1990) "Sharing memory robustly in message-passing systems," in *Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, PODC '90, ACM, New York, NY, USA, pp. 363–375.