Vasco Samuel Rodrigues Coelho

Bachelor of Computer Science and Engineering

Study and optimization of the memory management in Memcached

Dissertation submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Informatics Engineering

Adviser: Vitor Manuel Alves Duarte, Assistant Professor,
NOVA University of Lisbon

Co-adviser: João Carlos Antunes Leitão, Assistant Professor,
NOVA University of Lisbon

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

September, 2018

Study and optimization memory management in Memcached

This document was created using the (pdf)LaTeX processor, based in the "novathesis" template[1], developed at the Dep. Informática of FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

Abstract

Over the years the Internet has become more popular than ever and web applications like Facebook and Twitter are gaining more users. This results in generation of more and more data by the users which has to be efficiently managed, because access speed is an important factor nowadays, a user will not wait no more than three seconds for a web page to load before abandoning the site. In-memory key-value stores like Memcached and Redis are used to speed up web applications by speeding up access to the data by decreasing the number of accesses to the slower data storage's. The first implementation of Memcached, in the LiveJournal's website, showed that by using 28 instances of Memcached on ten unique hosts, caching the most popular 30GB of data can achieve a hit rate around 92%, reducing the number of accesses to the database and reducing the response time considerably.

Not all objects in cache take the same time to recompute, so this research is going to study and present a new cost aware memory management that is easy to integrate in a key-value store, with this approach being implemented in Memcached. The new memory management and cache will give some priority to key-value pairs that take longer to be recomputed. Instead of replacing Memcached's replacement structure and its policy, we simply add a new segment in each structure that is capable of storing the more costly key-value pairs. Apart from this new segment in each replacement structure, we created a new dynamic cost-aware rebalancing policy in Memcached, giving more memory to store more costly key-value pairs.

With the implementations of our approaches, we were able to offer a prototype that can be used to research the cost on the caching systems performance. In addition, we were able to improve in certain scenarios the access latency of the user and the total recomputation cost of the key-value stored in the system.

Keywords: Web cache; Memory management; Memcached; key-value stores

Resumo

No decorrer dos anos a Internet tem ficado cada vez mais popular e as aplicações web como o Facebook e o Twitter que têm cada vez mais utilizadores. Isto, resulta numa constante geração de dados pelos utilizadores que precisam de ser geridas eficientemente, porque a rapidez de acesso é cada vez mais um factor importante a ter em conta hoje em dia, um utilizador não espera mais que três segundos para que uma pagina web recarregue, antes de abandonar o mesmo. Repositórios em memória de chave-valor como o Memcached e o Redis são usados para acelerar aplicações trazendo os dados mais perto do utilizador, diminuindo o número de acessos aos repositórios de dados. A primeira implementação do Memcached no website LiveJournal, mostrou que usando 28 instâncias do Memcached em dez *hosts* únicos, fazendo *caching* dos 30GB de dados dos objectos mais populares conseguiu atingir uma taxa de sucesso de acerca de 92%, reduzindo o número de acessos para a base de dados e reduzindo consideravelmente o tempo de resposta.

Nem todos os objetos em cache têm o mesmo tempo de recomputação, desta forma esta investigação tem como objectivo fazer um estudo e apresentar uma nova abordagem de gestão de memória que seja de fácil de integração num repositório em memória de chave-valor, sendo a implementação desta abordagem feita no Memcached. A nova gestão de memória irá dar prioridade a chaves-valor que demorem mais tempo a ser recompudados.. Em vez de substituirmos ou modificarmos por completo a estrutura de substituição do Memcached e a sua politica de gestão de chaves-valor, nós adicionamos uma camada onde vão ser alocados os chave-valor com maior custo de recomputação. Para além desta nova camada, vamos também apresentar uma nova da política de rebalanceamento dando outra vez prioridade às chave-valor com maior tempo de recomputação, fazendo com que mais memória seja alocada a chaves-valor que tenham um grande custo de recomputação.

Com a implementação destas abordagens de gestão de memória conseguimos assim oferecer um protótipo preparado para investigar o custo da recomputação no desempenho dos sistemas de cache, melhorando também em certos casos o tempo de acesso do utilizador a aplicações web e reduzindo o tempo total da recomputação dos objectos que passam a ficar no sistema de cache.

Palavras-chave: Cache web; Gestão de Memória; Memcached; Repositórios chave-valor

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context Work and Motivation

Over the years the internet has become more popular than ever and the [7] number of users on it has increased considerably [19]. The World Wide Web is considered to be a large distributed information system that provides access to shared data objects. With its increase in popularity, the percentage of network traffic scaled over time. This popularity has proliferated, possibly because the Web is relatively inexpensive to use and accessing data through it is faster than any other means [40].

Over time there has been a substantial increase of users in social media networks [27, 38]. In the case of Twitter's users, an average of 58 million tweets is sent every day [37]. This results in millions of page request that the company needs to deal with high speed to satisfy user needs. Studies have shown that a user will wait no more than three seconds for a web page to load before abandoning the site [30]. The traditional disk-based storage systems are not enough to handle this situation due to the fast data access required by users, degrading the overall performance significantly [46]. Since most of these requests are static documents [7] (e.g. HTML pages, pictures, video and audio files), distributed caching can reduce this limitation by bringing the data closer to users, reducing the access latency of content and the number of accesses to the database. [45].

In-memory key-value stores are a type of data storage that is usually designed for read-heavy applications. They work similarly to a commonly known data structure: a dictionary. These in-memory key-value stores are used to implement distributed caches, and they have an essential role in today's large-scale websites [3]. Memcached [16, 25] and Redis [8, 28] are some popular open-source distributed key-value stores being used by large companies and communities. In the specific case of Memcached, it is used by Facebook, Twitter, Wikipedia and many others [16, 26, 36, 47]. Likewise, Github, Flickr,

Stack Overflow and others use Redis as their distributed caching system [15, 17, 29, 35]. The first implementation of Memcached [16], on LiveJournal's website, showed that by using 28 instances of Memcached on ten unique hosts, caching the most popular 30GB of data can achieve a hit rate around 92%, reducing the number of accesses to the database and response time.

When key-value stores are full of objects, and a new object is to be stored, older objects need to be evicted, i.e. they need to be removed to free space for other objects. Such evictions are decided through replacement policies. Memcached generally uses a trivial Least Recently Used (LRU) decision-making when it comes to evicting objects. Recency is highly used as a replacement policy in caches, and it evicts objects that are less likely to be requested, giving priority to data that is usually used. LRU works well for workloads in which recency gives good predictability on requests. However, like Zaidenberg et al. [44] said, a replacement policy should suit the workload and the infrastructure that is supposed to serve. Factors such as size and latency of the object have an important role in today's web applications. Web objects are not uniform, they vary in size, and we need to take this factor into consideration because it may be beneficial to store smaller objects that take less space in cache than bigger ones. Accordingly, latency also has an important role in web caching. Web objects may take much time to download or to compute, and it can be beneficial to give priority to objects that have more download latency. If an object has low latency, it is easier to recompute it. The replacement policy can then decide to evict these low latency objects, leaving more costly objects in the cache, providing users with better response when more costly objects are requested.

Some replacement policies combine the size and the cost of recomputing the object (download latency) [7, 22], solving the problem of making decisions for non-uniform objects. The GD-Wheel algorithm [22] is one of the replacement policies that can deal with non-uniform objects. It combines the Greedy Dual algorithm [7] and hierarchical cost wheels, presenting an amortized time complexity per operation. Nevertheless, these policies cannot be easily implemented in Memcached because Memcached presents a particular memory management scheme. This scheme makes Memcached to only perform well with policies that present recency as there decision factor. If other policies would be implemented in Memcached, most likely the memory management scheme would need to suffer a significant change to adapt to the new policy.

Memcached memory management is designed based on the slab allocator of SunOS 5.4 [6]. The memory is partitioned into classes, called slab classes. A slab class is composed of a group of slabs. Each Slab it is divided into equal size blocks of memory called chunks, these chunks store objects of a specific range size and range differ from class to class.

Memcached uses a rebalancing policy based on the eviction rates between slab classes. Rebalancing policies are used to balance slabs between the slab classes to avoid a problem called slab calcification – a problem that arises because most slabs were allocated to former popular slab classes and when those classes stop being popular, the newer more popular slab classes present a low number of slabs and fill up their capacity quickly,

resulting in high eviction rates in the latter classes, and a *calcified* state on the former, i.e. most of their memory isn't used.

Conglong Li et al. [22] replaced Memcached original rebalancing policy with his new cost aware rebalancing policy. This change was necessary in order for him to apply his GD-Wheel algorithm in Memcached. Nonetheless, he assumes that all of the results come from his GD-Wheel replacement policy, ignoring that his new cost aware rebalancing policy may influence on having better results. Some researchers [10, 12, 13] have stated that by using dynamic management between slabs, Memcached can achieve better results. We believe that by mixing these dynamic management algorithms with some cost aware rebalancing policies it can result in a better performance by increasing the hit ratio on cost aware replacement policies like the GD-Wheel algorithm. A.Cidon [13] referred that if we assume that the cache average read latency is 200μs and the MySQL average read latency is 10ms, increasing the hit rate by 1% would reduce the read latency by over 35%. This means that from 376μs at 98.2% hit rate to 278μss at 99.2% hit rate. If we manage to increase this 1% on the hit ratio by using a dynamic memory allocation algorithm with a cost-aware replacement policy, we can see that we achieve a major contribution allowing to reduce the applications read latency significantly.

1.2 Problem Statement

Cost-aware Replacement policies are arising to tackle systems that benefit from caching non-uniform cost based objects in their key-value stores. These policies are not supported in Memcached for efficiency reasons, this instance of a key-value store presents a specific memory management scheme, and for this reason, Memcached only works well with LRU policies. Cost-Aware replacement policies like the GD-Wheel were implemented into Memcached to prioritize more costly key-value pairs. Nevertheless, solutions like this require a big change in Memcached's management scheme to adapt to new cost-aware key-value pairs, leading to rough solutions with high complexity. Some cost aware rebalancing policies solutions like the Cost-Aware rebalancing policy [22] were made in the attempt to incorporate their replacement policy in Memcached. This solution allowed Memcached to rebalance its slabs taking into consideration the average cost of each class, making it possible to combine with the GD-Wheel algorithm. However, the authors do not explore in depth their rebalancing policy, ignoring the possibility that it may have some influence on their results. After an object is evicted the Cost-Aware rebalancing policy moves a page from the lowest priority queue to a higher priority queue in Memcached. We believe that by constantly moving the slabs from the lowest priority slab class might not always be beneficial. There could be some classes where the reallocation of their slabs does not increase the miss rate as much as if we constantly take from the low priority queue.

1.3 Objectives

Our objective is to study and come upon a cost-aware memory management solution that can be easily implemented in a key-value store, concentrating on the implementation of our solution in Memcached. By studying how cost aware memory management schemes can influence cost based replacement policies in achieving better results on Memcached, we can see the importance of this memory management when new factors of the objects are taking into account e.g cost and size. Our priority in this study is to show that the client can benefit more if Memcached prioritizes more key-value pairs to stay in cache if they take longer to time to recompute, giving this way faster responses to the requests. This research will also allow us to compare our proposed solutions with another Cost-Aware Replacement and Rebalancing Policy created by Conglong Li et al.

1.4 Contributions

We expect to present a model of a new system that can tackle the memory management of Memcached by turning it cost-aware. The new system would have an instance of Memcached running. We then would modify its default replacement and rebalancing policy. Since Memcached does not present a structure capable of managing its key-value pairs by their recomputation cost, we will modify the default replacement structure to a more cost-aware replacement strategy, capable of constraining the most costly key-value pairs in cache. We plan that this modification can be easily integrated into Memcached with the purpose of serving an easy to implement solution. Additionally, we will replace Memcached's rebalancing policy for a new dynamic cost-aware rebalancing policy. The new dynamic cost-aware rebalancing policy will give more memory for storing more costly key-value pairs and it will react when the hit-ratio starts to decrease. A prototype will be made to simulate the model of the new system that was planned. Consequently, an analysis and a comparison with GD-Wheel will be made from the gathered data provided by the prototype.

1.5 Document Organization

The next chapters of this document are organized in the following order:

- **Chapter 2**, presents the related work, first introduces the concept of key-value stores and next a bigger view of Memcached. After this, it focuses on approaches that can be done to optimize Memcache's memory management, replacement, and rebalancing policies.

- **Chapter 3** outlines our proposed solutions, for as cost-aware ready to use memory management for a generic key-value store and Memcached.

- **Chapter 4** describes the implementation of our proposed replacement and rebalancing policies in Memcached.
- **Chapter 5** describes our experimental environment and presents the results of our experiments.
- **Chapter 6** wraps up the work done in this thesis, presenting some conclusion of our work and some ideas for our future work.

Chapter 2

Related Work

2.1 Key-Value Stores

A key-value store as Joe Celko [11] describes in his book, is a collection of pairs of (<Key>,<Value>), that generalize a simple array. The keys are unique, which means the collection only have one pair for each key. The pairs can be represented by any data type that can be tested for equality. Key-value stores can support only four basic operations:

- **Insert**: stores a pair into the collection.
- **Delete**: removes a pair from the collection.
- **Update**: changes the value of the associated key of the pair
- **Find**: searches in the collection for the value of the associated key. If there is no such value, then an exception is returned.

Key-value stores are vastly used in the cloud community, as the simplest form of NoSQL databases and present an alternative to traditional relational database stores (SQL). This popularity is due to the benefits that key-value stores can bring on scalability that the SQL databases cannot provide. Unlike SQL databases, key-value stores do not necessarily need to have a schema, putting away all of the data integrity in the application. In other words, this means that key-value stores do not rely on formally defined data types, allowing any kind of data in the system.

SQL databases use a process called the normalization of the data. The normalization consists in organizing the data into small logical tables in a way that the results are always unambiguous and ensuring that the data dependencies are clear. This way the SQL databases avoid replication, redundancy, anomalies and can efficiently manage its

data, but in exchange, performance is lost when processing the data because sometimes the normalization process can become too complex and it will involve many tables to create a logical meaning. On the other hand, Key-value stores do not usually work with data relations, therefore normalization is rarely used in this NoSQL data model. Taking all this into consideration, key-value stores have a simple structure, and their query speed is higher than relational databases. The possibility of not having a schema or a normalization enables key-value stores to support distributed mass storage and high concurrency easily. Furthermore searching and modifying data operations are more efficient in key-value stores and can deal with high data values and disks, making key-value stores useful for high read-applications by scaling with more resources. Also Key-value stores can store its data in memory like RAM, or even in solid state drives and even rotating disks. Also it supports the use of eventual consistency models or serializabilitiy. As an example, for eventual consistent key-value stores it exists: Dynamo [14], Oracle NoSQL Database [31], Riak [21]; as for in-memory key-value stores: Memcached [16] and Redis [8].

2.1.1 Key-value Stores as a Web Cache Server

A web cache is a temporary storage that contains web documents and other data like HTML pages, videos and images. The point of using web caches is to reduce the latency of access to the databases or some other sources. Caches can contain a subset of data that is also stored in a database. The goal is to keep the most used web documents in the cache to avoid taking more time to the database or generating content from other sources. When cache fills, replacement policies are used, to replace no longer accessed data with the new data.

In-memory key-value stores[1] can behave like web caches. The data structure of a key-value store and its capability of storing massive amounts of data in a highly concurrent fashion provide the opportunity for applications like Amazon ElasticCache [42] to be optimised for read-heavy workloads. The data stored in the in-memory key-value stores may be intensive input/output from the database queries and it can also be objects that are computationally intense to calculate.

Typically In-Memory key-Value stores support two operations: GET and SET. The GET operation retrieves the value of the associative key and the SET inserts into the key-value store the key-value pair. In figure 2.1. we can see an example of a key-value store working as a web page cache. When the application receives an HTTP Request (step 1), it will generate a GET request (step 2) that is sent to the cache (step 3). The key-value store will lookup if there is a match for the key requested and if a match occurs, the key-value store will return the value associated to the requested key. Likewise, when the cache does not find what was requested, it will return a null value to the application (step 4). Depending on the result of the key-value store, if a value was returned, the application

[1] In-memory key-value refer to a key-value store that resides in central memory (RAM)

will jump to (step 7) and generates the response, returning the HTTP Response to the client (step 8). On the other hand, when a null is returned the application tries to access the database (step 5). The database will process the query and will return the result of the execution (step 6). The application then will generate the HTTP page and will return it to the client (step 7 & 8). After the generation of the response, the application can choose to update the new data in the cache (step 9).

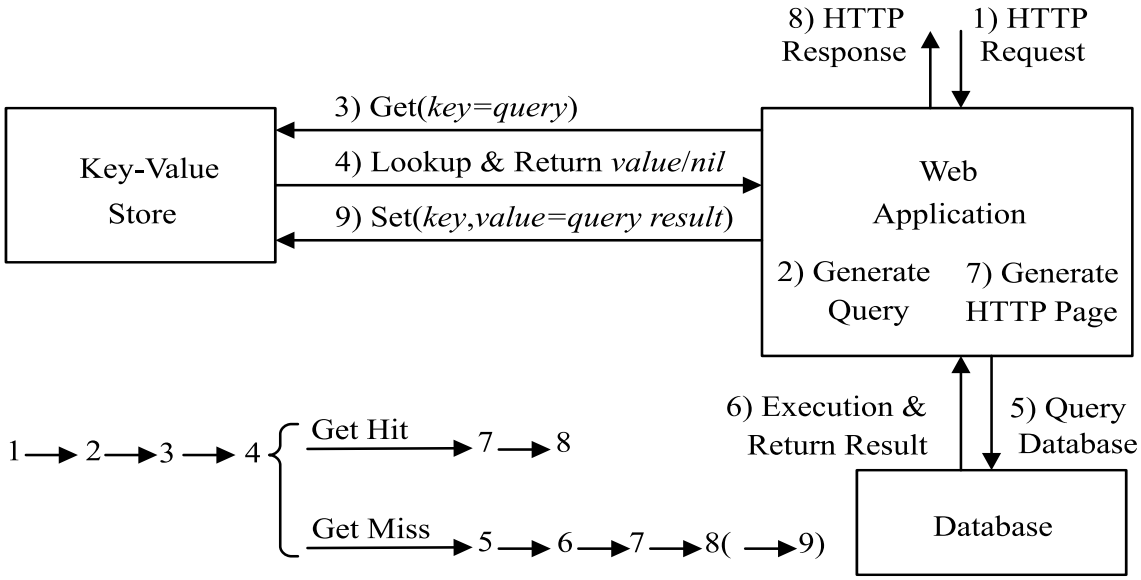

Figure 2.1: Using a Key-Value Store as a Database Query Cache.(taken from [22]).

2.2 Memcached

Memcached [16, 25] is a distributed in-memory cache supporting arbitrary chunks of data as its data type. It is multi-threaded, making it ideal for read-only data. Memcached presents to the user an optimised dictionary interface (key-value pair), storing each object in its memory. Several companies used or still use Memcached like Facebook, Twitter, Wikipedia and many others [16, 26, 36, 47].

As illustrated in figure 2.2, we can see an example of how Memcached works as a distributed memory caching system, deploying four Memcached servers to store its respective data. In step 1, an application will request for a key FOO and BAS to the Memcached Client. The Memcached Client is responsible for sending requests to the correct Memcached Servers. When receiving the application requests, it will hash [34] each key in order to know which Memcached Server should receive requests. In parallel, the Memcached Client will send the requests to the intended Memcached Servers as you can see from step 2. The Memcached Servers replies to Memcached Client, step3. In step 4, Memcached Client will aggregate the responses and send them to the application. From the example above we can see that Memcached does not provide redundancy but distributes load and achieves better performance. For each key it picks the same Memcache Server

consistently because each key is hashed in order to determine which Memcached server should handle the respective keys. We can see also that every instance of Memcached combined represents one big cache. Therefore each distributed Memcached is a fragment of the hypothetical big cache that is consistently dealing with the same set of keys.

Memcached relies on its fast memory management scheme to achieve his greatest local performance. We are going to briefly describe in the respective sections below, how each aspect of the memory management works in Memcached.

2.3 Memcached Components

In the light of what Conglong Li and Alan Cox said [22] there are three major components of Memcached, the hash table, the replacement structure and the memory allocator. The memory allocator is responsible for every memory allocation of a key-value pair in Memcached. The replacement structure is responsible for choosing which key-value pairs are going to be evicted, and the hash table is responsible for associating the hash value of the requested key to the place where the key-value pair is stored in memory. Every key-value pair has its metadata, information that tells us for example which are the pointers to the next and previous key-value pairs in a linked list, the expiration time of the key-value pair in cache, the respective key and value of the pair and many others. Thus, when a key-value pair is allocated in memory, what is actually is allocated is the key-value pairs and its respective metadata.

When a key-value pair is going to be stored in Memcached, first the memory allocator will check if there is enough memory to store the key-value pair. If there is no memory, the memory allocator tells the replacement structure to evict a key-value pair. After the eviction, the memory allocator reclaims memory to store the new key-value pair. The replacement structure is composed by a doubly linked list of key-value pairs that will pick the key-value pair to evict with the least priority.

In the case of Memcached receiving a GET request, the requested key is hashed and right after the hash table maps the key-value pair that associates it to the requested key. If the key-value pair is found, it will be returned to the client and the key-value pair is updated in the replacement structure.

2.3.1 Memory Allocation and Replacement Structure

Memcached, for its memory allocator, uses a technique called the Slab Allocator [6] that is described in more detail in section 2.4.3. Shortly the slab allocator is a memory management mechanism that is intended to avoid fragmentation. By using the slab allocator, Memcached starts organizing its memory in layers called slab classes. For every slab class, there is a limited size for storing an item. In figure 2.3 we see that the slab class 1 can take items up to 96 KBs, as for slab class 2 it can take items from 97 to 120 Kbs. This way, Memcached can efficiently manage its memory and reduce internal fragmentation. We

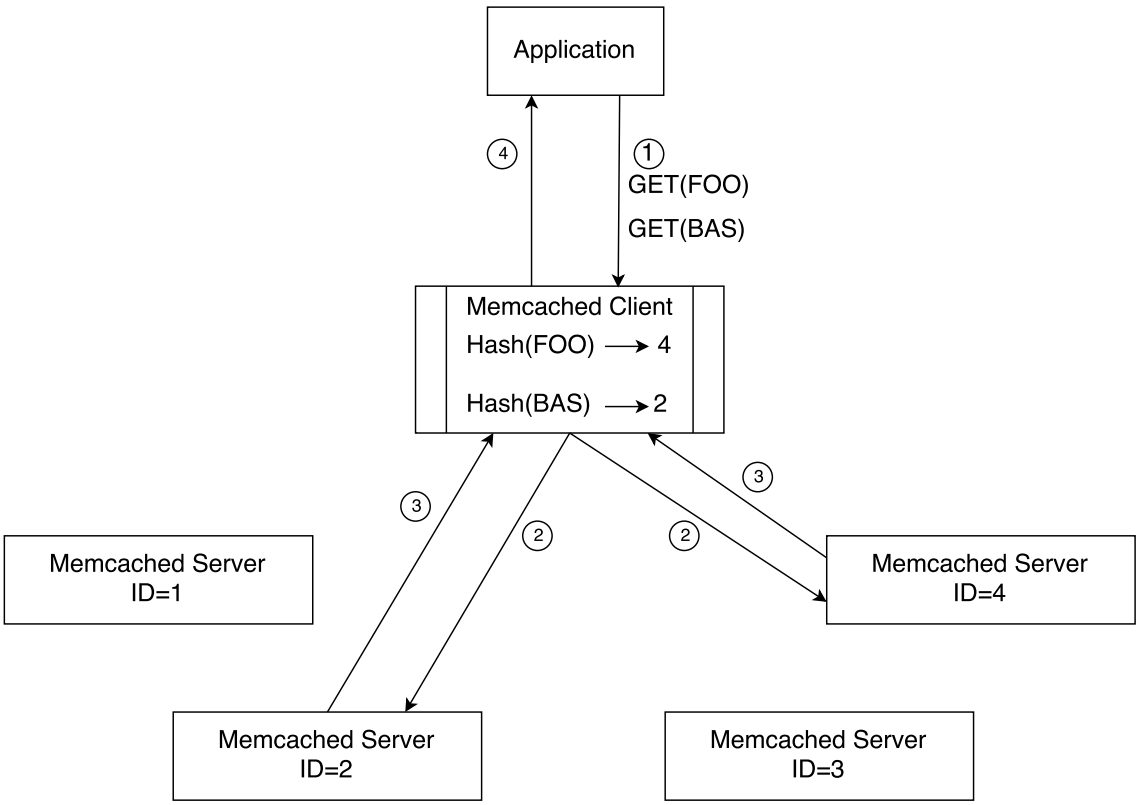

Figure 2.2: How Memcached works.

can see internal fragmentation as a block of memory assigned to an item that is somewhat larger than needed, leaving some portion of the memory unused by other items. Due to the fact of Memcached constraining the items into specific layers by their size, it can predict how much memory is needed to store an item. Even though the slab allocation technique cannot eliminate all internal fragmentation, it can greatly avoid a big part of it, allowing more items to be stored in Memcached.

To efficiently manage the items that will stay or leave the memory, we mention in section 2.3 that Memcached uses a replacement structure to do so. This structure is a doubly linked list and organizes its items by their recency, meaning that the least recent more popular items tend to be in cache. For every slab class, Memcached has one replacement structure, and the reason for this is that when an item is going to enter cache and memory is needed, Memcached can quickly know which is the least recent item available that will provide with sufficient memory for the newer one. This leads to fast management and allocation of the memory. We will provide more detail information about the replacement structure of Memcached in section 2.5.1.

2.4 Memory Management schemes

Operating System memory management focuses on handling the memory at two levels: i) allocating memory to the programs running in the operating system; and ii) the internal

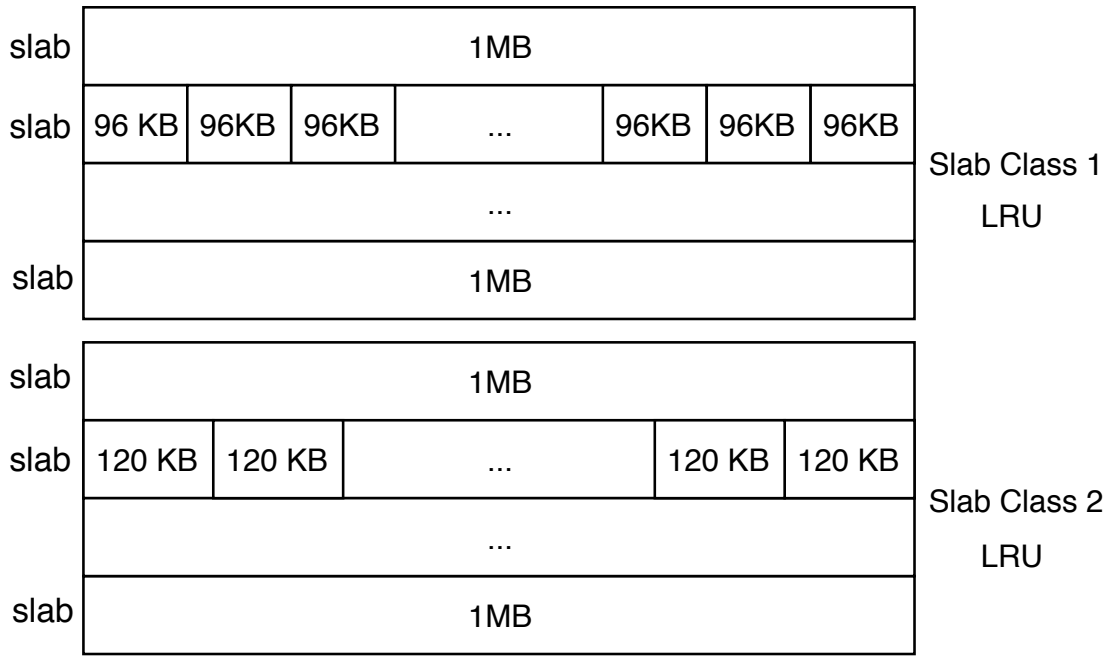

Figure 2.3: Example of the memory management of Memcached.

management dedicated to the kernel. In the first case for a program to execute, it is necessary for the actual program and its data are in memory, and for this reason, memory management handles the blocks of memory for such programs in the operating system in a way that contributes to optimizing the performance of the overall system. As for the kernel, the allocation and freeing of objects are common operations, therefore memory management needs to handle the memory efficiently and quickly, relying on its internal memory allocator to do so.

Web applications require most of the times to access their data in order to achieve fast decisions and to show more quickly what the user wishes. In-memory key-value stores are used to store the data more used, avoiding the intensive access to slower data storage. These in-memory key-value stores are frequently requested due to the needs of several web applications. Since many web applications need their data on the go, it is important that the in-memory of the key-value store has a good management of its memory, because the more data that can be stored, the less likely it will be necessary to access the database or remote resources.

In the following subsections, we are going to explore some memory management schemes that exist for in kernel and process management.

2.4.1 Fixed-size blocks allocation

A. Silberschatz et al. [32] said that the easiest method for allocating memory is to divide the memory into fixed-sized partitions. The Fixed-size memory allocation or fixed-partition scheme is mainly used in batch environments and has a straightforward approach when allocating its memory. The memory is divided into fixed blocks, and the operating system has a table which keeps track of the available and occupied blocks. Initially, all of the memory is available and when a process needs to allocate memory, the allocator will search for a block that is sufficiently large for this process. After finding a block big enough, the needed block is allocated, leaving the rest of the unused memory for future requests. Only one process can occupy a block so as an example if we have blocks of $5KB$ and two processes request to allocate $4KB$ and $1KB$ respectively, it will lead to using two $5KB$ blocks and $5KB$ are lost with internal fragmentation. When a process terminates, it releases its memory, allowing other processes to allocate that block. Now let us imagine that there are free size blocks of memory and five $1KB$ objects can be spread in memory. After those objects are freed, it will fragment the memory, causing a $5KB$ object request not to be satisfied. Although the total free memory is enough to allocate the process of $5KB$ since it is not contiguous, it can not be used, causing a problem called external fragmentation.

Eventually, the memory will be fragmented, and the processes that are in need of memory must wait in a queue for there turn for some continuous blocks of memory to be available. The allocator needs to know from the set of free blocks which block best satisfies the request of size n. There are some strategies to tackle this problem in which the first-fit, best-fit and worst-fit are the most commonly used.

- **First fit**. The allocator will allocate the first large enough block that he will find for an in need processor. The disadvantage of using this strategy it is the possibility of a big internal fragmentation occur inside blocks. A processor that needs $200KB$ and the first large enough block suited for the processor is $600KB$ the internal fragmentation will be $400KB$. Nevertheless, this is generally the faster strategy because we allocate the processor in the first suitable block.

- **Best fit**. Using this strategy, the allocator will pick the smallest big enough block that the processor could fit in. To do so, the allocator checks the whole list of available blocks, unless the list is ordered by size. Usually, we waste some performance since the whole list of available blocks is iterated, but on the other hand, this strategy produces the smallest internal fragmentation

- **Worst fit**. This strategy iterates the whole list of available blocks and will find the largest big enough block that the process could fit in. It works similarly like Best fit, but it will produce the biggest internal fragmentation possible. A. Silberschatz et al. [32] refer that simulations have shown that First fit and Best fit are better than worse fit in decreasing time and storage utilization.

The fixed-size management scheme allocates fixed-size blocks of memory continuously and with the allocation and freeing of the memory it will eventually lead to internal and external fragmentation. However, some memory management schemes allow the allocation of the blocks scattered in memory like paging.

Paging memory allocation allows the physical address space of a process to be non-contiguous, avoiding the problem of fitting memory chunks of fixed-size onto the backing store.

There are some ways of implementing paging like shared pages, hierarchical pages, inverted pages and others, but we are going to focus more on the basic method for implementing paging. The basic method for implementing paging consists of breaking psychical memory into fixed-sized blocks called **frames**, also the virtual memory will be broken into blocks of the same size called **pages**. The hardware defines the size of the pages and the frames have the same size as the pages. Each virtual address generated consists in two parts: **page number** and a **page offset**. When processes request memory, they first need to allocate the pages on the virtual memory, for an example if one page is $4MB$ and a process needs to allocate $6MB$ then two pages will be allocated to that process. Even though a process does not completely fill a page, it is still allocated to that same process, making it unusable to other processes and creating internal fragmentation. **Page table**, as we can see from figure 2.4 contains the base address of each page in physical memory. Each index represents the number of the page and each page allocates a frame, together with the offset of the page, it is possible to determine where is the page going to be stored in the physical memory.

In the next example based on A. Silberschatz et al. example 2.4 we are going to describe how pages are assigned to physical memory. Let's Consider that a page is 4 bytes and the physical memory is 32bytes. The virtual address 0 is page 0 and offset 0. Since the page table indicates that the page points to frame 5, it is going to be mapped as follow: $((5 * 4) + 0) = 20$. The equation represents the frame associated to the page times the bytes of a page plus the offset of the page. For page 12, the physical address that is going to be map is 8 because $((2 * 4) + 0) = 8$.

2.4.2 Segmentation Memory Management

Segmentation is a commonly used memory management that is still used today. A logical address space is a collection of segments. Segmentation can be combined with virtual memory and even paging, but in this subsection, we are going to focus on simple segmentation that is not combined with either the above memory management techniques.

Segmentation is a variant of dynamic partitioning where a process is broken into segments. This approach of memory management is visible to the programmer, on the contrary of paging that is entirely handled by hardware and the operating system. A single process is commonly split into three segments: data, code, and stack. The data is where all the variables used in the program; the code is part of the process that is

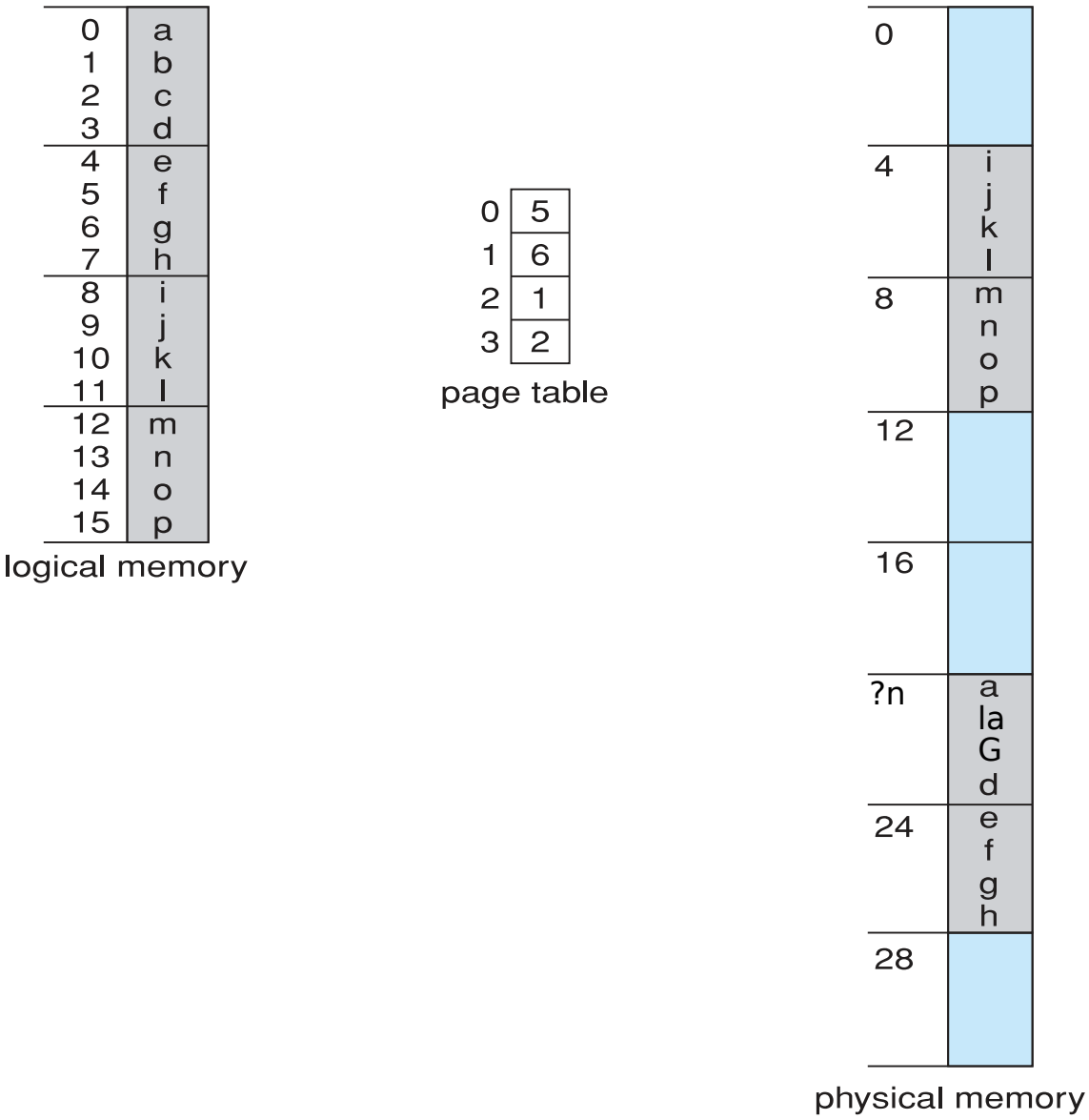

Figure 2.4: Paging model of logical and physical memory. (taken from [32]).

actually executed on the processor; Stack that dynamically tracks the progress of the code. Processes can have more segments, and usually, more complex objects have segments for objects. Segments vary in length and the length of a segment is defined as its purpose in the program. Each segment has a name or number and a length.

Although the user can represent memory with two-dimensional address, memory needs to be mapped to physical memory. To do this, a **segment table** exists to know each segment base and the segment limit. The segment base tells where the segment starts in the physical memory, and the segment limit is the length of the segment. The index of the table consists of the number of the segment. The offset of the segment must be between 0 and the limit of the segment.

As an example, if we look to figure 2.5 we can see the segmentation table that points

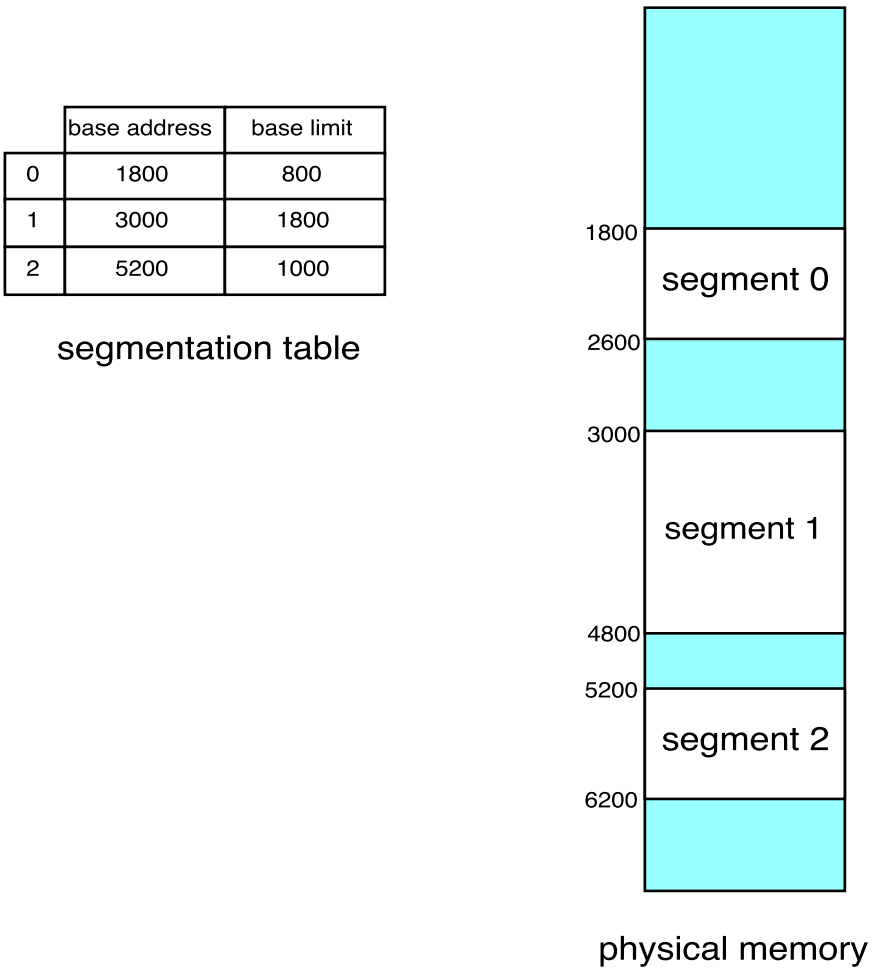

Figure 2.5: Example of segmentation. (inspired by [32]).

to three segments from 0 to 2. The segment 0 begins at location 1800 and as a limit of 800 bytes. Therefore a reference to byte 160 of segment 0 is mapped to location 1960 since $1800 + 160 = 1960$. However, a reference to 1400 bytes of segment 2 would result in a trap to the operating system, which means the logical addressing attempt is beyond the end of the segment.

2.4.3 Slab allocation

The slab allocator [6] is an object-caching memory allocator used in the kernel and also in some processes malloc/free implementations presented by Jeff Bonwick in 1994. Allocating and freeing objects in memory are common operations in the kernel. When creating an object, first its memory must be allocated before proceeding with its initialization. After an object is no longer needed, it is destroyed and memory is freed. It is expensive to create and destroy objects and also it can increase the fragmentation of the memory. Therefore slab allocation addresses the problem by caching these objects in its free or in-use state, avoiding the frequent allocation and freeing of objects in memory.

In SunOS 5.4, the Slab allocator works with units called slabs. A slab represents one

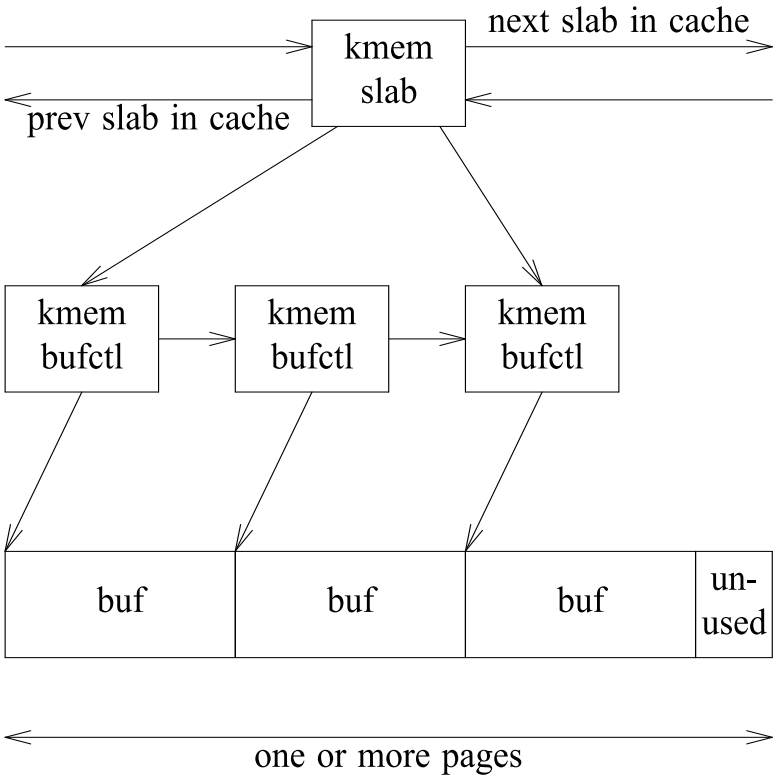

Figure 2.6: Slab logical layout. (Taken from [32]).

or more pages of virtually contiguous memory in which each slab is split into equal sized chunks. The slabs are in the kmem_slab data structure. Like its represented in figure 2.6 we can see that each kmem_slab holds the information of the slab's linkage in the cache, the reference count and the list of free buffers. The kmem_bufctl structure keeps the free list linkage, the buffer address and a back-pointer to the controlling slab. The slabs are in a circular doubly-linked list, in which the list is partially sorted. First on the list comes the empty slabs (all buffers allocated), then it follows by partially filled slabs(some buffers are allocated and some are free), and then at the end of the list, there are the complete slabs (all buffers are free). The cache contains a free list pointer that points to the non-empty slabs and each slab has its own free list of available buffers. This originates a two-level free list structure, that helps the memory reclaiming because it just simply needs to unlink a slab instead of unlinking every buffer from the cache's free list.

The slab data structure as J.Bonwick [6] said, can bring several advantages when managing the memory:

- **Reclaiming unused memory is trivial.** As we mentioned above, the cache presents a two-level free list structure, this makes the process of reclaiming unused memory straightforward. By simply putting the slab reference count to zero, the associated pages can be returned to the VM system.

- **Allocating and freeing memory are fast constant-time operations.** To perform

these operations we only need to remove or add objects to the free list and update a reference count. Since the free list is implemented with a doubly-linked list, it takes constant time removing or adding objects from the list.

- **Severe external fragmentation (unused buffers on the free list) is unlikely.** The slab allocator pre-allocates its memory and this way, the external memory is controlled. For a sequence of 32byte and 40byte, the biggest external fragmentation is 8byte since we can only allocate 32 byte and 40byte.
- **Internal fragmentation (per-buffer wasted space) is minimal.** Each buffer is exactly the cache object size, therefore the only wasted space is an unused portion at the end of the slab. As an example, a 4096 byte pages and the slabs in a 400 byte object cache would make each slab to have ten buffers, leaving an unused 96 bytes. This results in 9.6 bytes wasted per slab, that represents a 2.4% internal fragmentation.

Memcached inspires in the slab allocation technique for its memory management. In figure 2.3 we can see three important components presented in Memcached, similar to the concepts presented above:

- The **slab** which it is the primary unit of currency, representing blocks of memory that usually are $1MB$ each.
- The **chunks** are the result of slicing up a slab. It is in the chunks where the key-value pairs are stored. Every chunk of a slab as the same size.
- The slabs are stored in partitions of the memory depending on their chunk size, these partitions are called **slab classes**. In essence, slabs that are in different Slab classes contain chunks of different size.

Memcached initially starts by dividing the memory into slabs. The total available memory is allocated in the form of slabs into the slab classes. When a slab is allocated into a slab class, it divides it into chunks of a specific size. The chunks of slabs increase between slab classes by a default factor of 1.25. To know which slab class a key-value pair is going to be stored, Memcached checks the size of the key-value pair and stores it in the slab class with the smallest chunks sufficient to store the key-value pair. If we take as an example figure 2.3, when a key-value pair of $92KB$ is being inserted in Memcached, it will be inserted in Slab Class 1 because it has chunks sufficiently large to store the key-value pair. Hypothetically the key-value pair could be stored in slab class 2, but it would have caused a bigger internal fragmentation of $28KB$.

2.4.4 Summary

Memory management schemes were built to handle memory in a way that processes can reach an optimized performance. Like Fixed-Size block allocation, it ensures that the

memory should be allocated in a continuous fashion way, partitioning the memory in various fixed size blocks. This method can bring some disadvantages. The continuously fixed size partition brings internal fragmentation in each block. Occasionally processes request blocks of memory, and most of the times the allocations are bigger than needed, leaving remaining free memory unused.

Paging addresses the memory in a non-continuous way, avoiding the problem of fitting memory chunks of various sizes onto the backing store. It partitions the physical and virtual memory into fixed-size blocks called frames and pages, respectively, reducing external fragmentation and the internal fragmentation. Thanks to the page mapping system, pages are allocated anywhere in the physical memory. The page management is not visible to the user and is controlled by the operating system.

Segmentation, on the other hand of paging, it is a memory management scheme that is visible to the user and segments do not follow a specific fixed size. Segmentation works similarly like paging. It also has a segmentation table, which indicates where the segmentation starts and ends. Any process can allocate a segment if there is available memory because a segment does not follow any specific size. For this reason, segmentation avoids internal fragmentation. On the other hand, it has more external fragmentation because freed segments leave empty spaces that fragment the free memory space. Also, since the memory is not partitioned it takes more costly memory management algorithms to allocate segments. Paging and segmentation can be combined to solve problems like the segmentation, external fragmentation and simplifying memory allocations.

All memory management schemes described were generic types of memory allocation. Slab allocation is an object-caching kernel memory allocator that manages objects within the cache. The way slab allocation was designed shows that allocating and freeing objects in cache takes constant time to perform. This is a major advantage because it is common to insert and to delete objects in the cache. Furthermore, the Slab allocator reduces internal fragmentation by pre-allocating most of its memory, at the same time, it can also control its external fragmentation by designating a range of objects sizes to be stored in a particular area of the memory.

2.5 Replacement Policies

When the cache becomes full of objects and it wants to store new objects, it is performed a replacement policy. The replacement process is performed by evicting older objects in order to make space for newer ones. This way, the cache can adapt to the change of new requests pattern achieving better hit rates. To decide which objects are going to be removed, replacement policies take into consideration some factors of the objects. Krishnamurthy and Rexford [41] describe in their book these factors that can influence the decision of replacement policies as:

- **recency**: it is the time since the object was last accessed.

- **frequency**: number of times the object was requested or accessed.
- **size**: size of the object.
- **cost**: this cost refers to the latency of fetching an object from the original source or recalculating the object.
- **modification time**: last time the object was modified.
- **expiration time**: time defined for the object to be replaced (this is mostly used as an ageing process to remove objects that were popular once but are not anymore).

There are different replacement policies derived from these factors. Some even combine some factors of objects to cover the flaws of other used policies.

Recency aware policies. Most strategies tackle recency by using extensions of the Least Recently Used (LRU) policy [24]. The LRU policy assumes that the least recently accessed object has the least probability to be accessed in the future. The advantage of using this algorithm is that incorporates an ageing mechanism in objects e.g if an object stops being so popular [2] for a period of time, other recent objects that are starting to get more popular will have more priority of not being evicted. This way, objects that were once popular will eventually be evicted from the cache, giving a chance to more recent stored objects to scale their priority in the cache. The disadvantage of using LRU is that sometimes there are popular objects that can collide with recent objects. The recent ones have more priority because the eviction decision is done within a period of time where the recent objects were more accessed than popular ones.

Frequency aware policies. Frequency policies like The Least Frequently Used (LFU) policy [24], consider frequency when evicting an object. On the contrary of the LRU, LFU ignores recency on eviction and removes the object which has the least hits in the cache. By Ignoring the recency of the object, it can bring some disadvantages to the LFU policy, for example, new objects will not have time to build up their frequency to match older objects. Also, an object that was popular once and received many hits may never leave the cache even if it is no longer popular.

Size aware policies. Size is another factor used when evicting objects from the cache. For example, larger objects can be removed first to make room for more smaller objects. The advantage of this technique is that by removing larger objects, smaller objects can fit in the space left by the larger object. However, some websites can benefit from larger objects because some of them are harder to compute and therefore, it is better to remove the smaller objects first. Since we have the dilemma "Is bigger better?", the size factor is usually combined with other factors like recency, frequency or cost: Greedy Dual [7], LRU-Min [1], PSS [2].

Cost aware policies. Web objects are constantly being recomputed. Some objects, in order to be recompute need to gather information that is stored in databases. The latency

[2] popular objects are refereed as the objects that are more accessed since the time their are in cache

cost of going to the database and computing the object to deliver to the client can vary in most objects. This cost of recomputing can be essential in some systems. Since most caches use recency as their deciding factor, the cost is usually combined with recency and other factors to prioritize more costly objects to stay in cache. Replacement policies like Greedy Dual [7] and GD-Wheel [22] combine the cost of recomputing objects, size and recency to decide which objects should be evicted from the cache.

Modification time aware policies. When modification time is taken into consideration, what is done is to consider the period time since the object was last modified. Although, we can also consider as the eviction decision the last time since the object was accessed. The major disadvantage of this policy is that the objects that are frequently popular can be evicted because the time since the object was last accessed is longer than the time of the other less requested objects. Instead, this factor is used as an ageing mechanism and it is combined with other factors to tackle their disadvantages, like the frequency factor. A policy like Hyperbolic Caching [5] is an example that combines frequency and modification time. They use a function for ranking each object in cache, and Hyperbolic Caching based on that rank makes their eviction decisions.

Expiration time aware policies. Some caches have the chance of choosing the possible expiration time for evicting each object. This factor is not directly used in policies. When the object is not evicted by the replacement policy and its time in cache is longer than the defined expiration time, the cache forces an eviction on those objects.

In the following subsections, we are going to first describe in more detail how does the replacement policy works in Memcached, and after, we are going to explore some replacement policies that could benefit from cost factors.

2.5.1 Memcached Replacement Policy

When it comes to decisions to evict key-value pairs, Memcached uses a trivial LRU algorithm to pick the key-value pair that is ready to leave [44]. In this context, when we say that the key-value pair is least used in the cache, we mean that the key-value pair is the least recently requested among the other key-value pairs.

In Memcached for every slab class, there is a replacement structure that uses a LRU (Least Recently Used) algorithm as its replacement policy. Every replacement structure is a doubly linked list that can be segmented into three segments: Hot, Warm and Cold LRU. When a key-value pair is inserted, it goes straight to the Hot part of the LRU, giving it a maximum priority to stay in cache. After a time if the key-value pair is not requested often, Memcached sends it to the cold part of the LRU. If a key-value pair is requested more than 2 times in a period of time, it will become an active key-value pair. Active key-value pairs that are in the Hot segment will move to the Warm segment, active key-value pairs that are active in the Warm segment stay in the Warm segment, and finally, if active key-value pairs are active in the Cold segment, it will move to the Warm segment. The cold part of the LRU is the place where key-value pairs are not requested

often, and the ones at the bottom of the Cold LRU have the potential to be evicted. To manage the transition between this segmentations, Memcached uses a LRU maintainer. This maintainer will roam through every slab class, and checks in each segmentation of the LRU queue for key-value pairs that have conditions to change between segmentations or even to be evicted.

2.5.2 Greedy Dual-Size Algorithm

Greedy Dual-Size Algorithm [7] integrates recency of access with the cost and size of cached objects when making eviction decisions. P. Cao and S. Irani. introduced a Greedy Dual-Size Algorithm [7] with a more reduced time complexity than the original Greedy Dual from Young et al. [43]. Their solution uses a priority queue to store every cached object. To calculate the priority value H of an objects lets consider that the queue has a global inflation value L, which is added to newer items to allow them to have more priority from older ones. Let also $c(p)$ and $s(p)$ be respectively the cost and the size of the object p. If the object p is already in the memory, it will update the objects $H(p)$, by setting $L + \frac{c(p)}{s(p)}$. When the object is not in memory, and there is no more room to store it, the global variable L is updated, and it is set with the lowest $H(p)$ in the queue. After the update on L, the object with the lowest H is evicted and p is stored with the priority of the new $L + \frac{c(p)}{s(p)}$. If a key-value store chooses to use Greedy Dual-Size, the insertion or the access of an object in the key-value store takes logarithmic time to perform($O(\log n)$.) which will affect a lot the performance of the key-value store.

2.5.3 GD-Wheel Replacement Policy

Conglong Li et al. [22] said that if the cost of recomputing cache results varies significantly, then a cost-aware replacement policy will improve the web application performance.

They also stated [22] that since the GreedyDual requires a priority for each object, building an implementation based on GreedyDual that achieves constant time complexity seems almost impossible, however, if it is possible to restrict the priority range, amortized constant time complexity is achievable. They reduce the time complexity by leveraging the Greedy Dual algorithm with Hierarchical Cost Wheels that were inspired by Varghese and Lauck's Hierarchical Timing Wheels [39]

The Hierarchical Cost Wheel is a structure with a series of single cost wheels. A cost wheel is an array of queues with a clock hand pointing at one of his queues. We can see a clock hand as a pointer to a queue inside of a cost wheel. The time complexity of this should be logarithmic, but since they restrain the priority of the Hierarchical Cost Wheels, this does not happen. A cost wheel supports k different priorities, with k being the number of queues. When an object is inserted, if the clock hand is pointing at x queue, the object is placed into the $((c+x) \mod k)$ queue where c is the cost of the object. When there is no

room and objects need to be evicted, the clock advances until he finds a non-empty queue. After this step, he will remove the tail from the non-empty queue until the new object is allowed to be instantiated. The new object will be inserted into the head of the queue $((c+x) \mod k)$ which c is the cost of the object and x is the current position of the clock. Since a single cost wheel of size k supports up to k cost. Conglong li et al. efficiently extended the costs along the hierarchical cost wheels. Their fixed number of cost wheels is in a hierarchy that each higher level of cost wheel will support a bigger range of costs. Hierarchical Cost wheels act like single cost wheels, and objects are inserted or moved to the respective queue by adding the cost of the objects plus the current position of the clock hand. Evictions occur in the lowest level cost wheel and not in the higher cost wheels. The clock hand keeps moving until it finds a non-empty queue. When the clock hand completes a whole round, the clock hand from the higher cost wheel advances to the next queue. For example, if the clock hand of the first level cost wheel with size k moves the k positions finishing this way a whole round, the clock hand from the second cost wheel will advance and point to the next queue. After the clock advances in the higher cost wheel, migration is performed.

A migration consists in moving the objects of the queue that the clock hand of the higher cost wheel is pointing to, to the corresponding queues in the lower cost wheel. Since the objects are moving to a lower cost wheel, their priority needs to change. Let consider $c(p)$ to be the cost of the object p, $NQ1$ the number of queues of the higher cost wheel, $NQ2$ the number of queues in the lower cost wheel, $C[idx1]$ the current position of the clock hand in the higher cost wheel and NQ the number of queues in each cost wheel.The new priority queue in the lower cost wheel that the objects will be stored is determined by: $(\frac{c(p) \mod NQ1}{NQ2} + C[idx1]) \mod NQ$.

With this technique, they achieve amortized constant time complexity per operation if using a limited priority range. Since each queue is a doubly linked list, the insertion and reuse of objects take $O(1)$ time. Advancing the clock hand takes constant time because the queues are limited. With this, the eviction takes constant time to take an object in the lowest level cost wheel. If a migration happens, in the worst case, all cached objects are in that queue, and it will take $O(n)$ time. Nevertheless, migrating an object implies removing it from the higher cost wheel and inserting it in the lower cost wheel. These operations take constant time to perform. Taking into consideration the sequence of operations for migrating the objects, the time required to execute such operations is amortized constant time.

2.5.4 Summary

There are many factors that we can take into consideration when choosing the best replacement policy to optimize the cache of a given system. Some of these policies are combined in an attempt to optimize more complex systems. Although many systems use LRU as their replacement policy, we saw that some more complex systems could benefit

if they also took into consideration the cost of latency as their main factor of decision. Few replacement policies use cost as their deciding factor for evicting objects.

Algorithms like the Greedy Dual[7] and GD-Wheel[22] try to use the cost factor by combining other factors such as recency and the size of the object. Taking a more in-depth look into the Greedy Dual, we conclude that it uses a global inflation variable to simulate the recency factor. For each object, the algorithm assigns a priority through the inflation variable based on the object cost and size. All of the objects are stored in a single priority queue. Since every object is stored in a single queue, the time complexity for inserting and accessing an object is $O(\log k)$ where k is the number of objects. Key-value stores do GET and SET operations with constant time complexity, and therefore, by incorporating the Greedy Dual it might not be as beneficial as expected. However, GD-Wheel emerged to tackle this problem. Conglong Li et al. combined Greedy Dual and Hierarchical Cost Wheels, distributing objects through many hierarchical wheels. By limiting the hierarchical wheels, GD-Wheel can achieve constant amortized time complexity per operation. This makes it the best cost-aware replacement policy to choose when the latency is an important factor in the system.

2.6 Rebalancing Policies

Recapping from section 2.4.3, Memcached allocates its memory in the form of slabs, according to the requests distribution, some slab classes will have more memory than others. In a slab class that is more requested, it will have more memory allocated than the other less requested slab classes. Once the available memory of Memcached is assigned to a slab class, it will always remain associated to it. This means that when the request distributions changes in Memcached, some slab classes may not have enough memory to adapt to the new distribution and their replacement structures may be too short, forcing key-value pairs to be evicted too many times lowering the hit-ratio. This happens because the slabs were allocated taking into account the old request distribution, translating in a problem that is called slab calcification. If the distribution of the requests stays uniform, the hit ratio will not change much, and consequently the performance will be similar. However, there are systems whose request are not uniform and change over time, which will degrade performance. [9, 18]. To tackle the slab calcification problem, some approaches can be performed.

Cache reset. Every X seconds, all objects are removed from cache. Cache resetting is a manual policy and its not implemented in Memcached. This approach can bring several disadvantages[9] such as leaving client requests hanging, several periods of times to refill the cache with objects, and since the cache is empty, the database will receive many requests.

Rebalancing Policies. These type of policies will auto move slabs between slab classes like in the figure 2.7. The policies conditions restrict the movement of slabs to other slab

classes. As an example, Twemcache[3] [36] has a set of rebalancing policies to avoid the slab calcification.

- Random slab eviction: For every SET operation, if there is not any free chunk or slab in the slab class, Twemcache will randomly pick a slab from any slab class and will remove all objects. Accordingly, it reassigns the empty slab to the class in need by dividing the slab into chunks of the appropriate size.
- Least Recently Accessed Slab: Instead of picking a random slab to evict all objects, Twemcache picks the least accessed slab and removes all its objects reallocating it to the in need class.

Rebalancing policies are a good approach to confront the slab calcification problem. They are dynamic, and if we can detect when the slab calcification occurs as well which the classes will benefit and be harmed from removing slabs, we might reach a near optimal solution to the slab calcification problem.

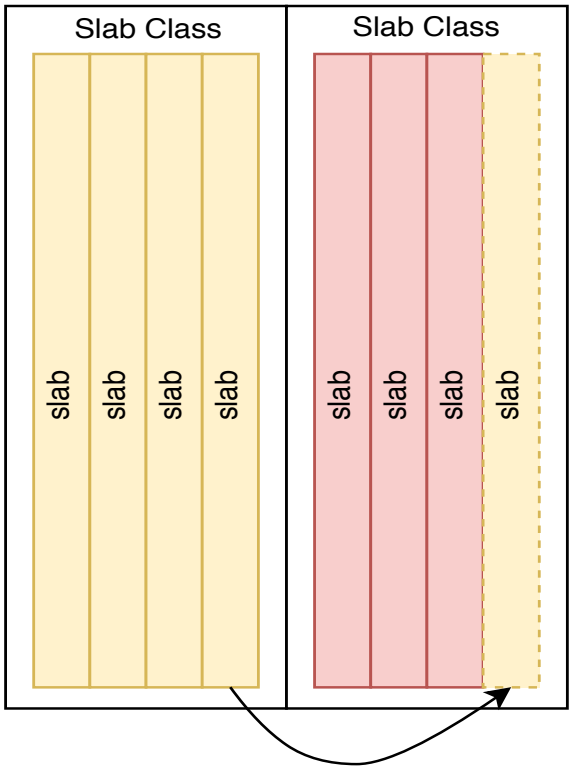

Figure 2.7: Example of a rebalance of slabs.

2.6.1 Memcached Rebalancing policy

Memcached rebalancing policy will check for the eviction rate of each slab class 3 times per 30 seconds. Once in every 10 seconds, if a slab class has the highest eviction count for

[3]Twencache is a variant of Memcached developed by Twitter

three times, it shows that the slab class is more likely to be the most requested of all slabs and for this reasons, it will take one slab from the slab class which had zero evictions. This approach has some inconsistencies, Conglong Li and Alan L. Cox, said that "this policy is very conservative and ineffective" [22].

One problem with this policy is that periodic decisions might be too lazy for fast object requests. Taking into consideration bigger corporations that deal with millions of requests from users per second, waiting 30 seconds for a cache to rebalance their slabs can have a significant impact on their performance. Furthermore, it could be beneficial to move slabs from a slab class with more chunks and a lower eviction rate to a slab class with fewer chunks and higher eviction rate, decreasing the miss rate.

2.6.2 Cost-Aware Rebalancing Policy

Conglong Li and Alan L. Cox [22] present in their paper an alternative to the original rebalancing policy of Memcached. The alternative consists of a cost aware rebalancing policy. Each slab class holds the average cost per byte information. A particular variable was created in Memcached to know which id of the slab class has the lowest average cost. Their policy reacts immediately on eviction. Every time an eviction occurs in a higher slab class, some least recently used slabs are taken from the lowest slab class. The number of slabs that are going to be moved depends on the size of the evicted key-value pair.

One problem with this policy is that it moves all slabs from the lowest slab class. At some point, the lowest slab class can have a lack of slabs. Also, since every slab is taken from the lowest slab class, lower objects will not have time to build up to migrate to other classes. Additionally, by rebalancing a slab on every eviction it can be a demanding process and can be inefficient if the distribution of the objects stays the same.

2.6.3 Cost-based Memory Partitioning and Management in Memcached

When using web caching, a big amount of objects are stored in the key-value store. These objects may not only have different sizes, but they may also have different retrieving costs. To deal with this, eviction policies like the Greedy dual algorithm [7] were developed. Nevertheless, solutions like this are not supported by Memcached because Memcached has a particular memory management scheme that works specifically with LRU policies. Introducing these solutions in Memcached would affect its performance. In section 2.2, we refer that Memcached partitions memory by the size of its objects. It means that when allocating an object if there is free space, it will first check the object size and allocates it in the memory dedicated to its size. When that part of the memory is full, it will evict one or more objects to make room for the new object.

Carra and Michiardi presented a solution [10] to allow cost-aware solutions to properly work with Memcached memory partition by tackling the memory management of Memcached. They propose an algorithm that evaluates a single slab movement from the class with the lower risk of increasing the number of misses, to the slab with the largest

risk of increasing the number of misses. For every slab class i, they consider the risk as the ratio between the cost of the misses due to eviction (mi) and the number of slabs (ri) allocated to that class (si): $\frac{mi}{risi}$. In essence, they evaluate the risk in every slab class and for every M misses they take a slab from the class which has the min $\frac{mi}{risi}$ and give it to the class with more mi. They can distinguish the misses from the objects that were not requested before from the objects that were already evicted by using a technique mentioned in [20] that uses two bloom filters ($b1$ and $b2$). When a SET operation is performed in the key-value store, it is possible to check if the key of the key-value pair is in both of the bloom filters; if this happens, the object is then registered as miss due to eviction. Likewise, the object is stored in $b1$. Periodically $b2$ is reset, the content of $b1$ is copied to $b2$ and $b1$ is reset. This approach is made to avoid the saturation of the bloom filter.

Their solution shows that the memory allocations adapts to the characteristics of the objects that are requested, thus obtaining a sub-optimal performance comparing to the solutions that statically allocate the memory [10].

2.6.4 Cliffhanger

In an attempt to solve the slab calcification problem A. Cidon et al. present Cliffhanger [13]. Cliffhanger is a lightweight iterative algorithm that runs on memory cache servers, and it performs a hill climbing algorithm in the hit-ratio curve to determine which slab classes may benefit from new memory.

To obtain the hit ratio curve Cliffhanger starts by running across the multiple eviction queues of Memcached and for each eviction queue, it determines the gradient of the hit rate curve at the current working point of the queues. In this process **shadow queues** are used instead of eviction queues to approximately determine the hit curve gradient. Shadow queues are queues that extend the eviction queues containing only the keys of the requests. A. Cidon et al. prove that although Cliffhanger is incremental and only relies on local knowledge of the hit rate curve, it can perform as well as a system with knowledge of the entire hit rate curve.

Cliffhanger with the knowledge of the local hit curve will incrementally allocate memory to the queues taking into account which one will benefit more from increasing their memory and decreasing the memory from the ones who benefit less. To do so, the process is as follow: i) If the request hits a shadow queue then its size will be increased by a constant credit; ii) Then it's randomly picked a different queue out of the list of queues that are being optimized; iii) After picking a queue its size is decreased by the constant credit; iv) When queues reach a certain amount of credits, it is allocated additional memory at expense of another queue.

The Hill climbing algorithm works well as long as the curve stay concavely and do not experience performance cliffs. Performance cliffs are regions on the hit rate curve where increasing the amount of memory can result in a drastic increase in the hit rate curve, removing the concavity of the curve. To overcome this, Cliffhanger uses a technique

inspired by the Talus algorithm [4]. Talus allows achieving the hit rate by calculating the linear interpolation between two points. In 2.8, you can see an example of how Talus works.

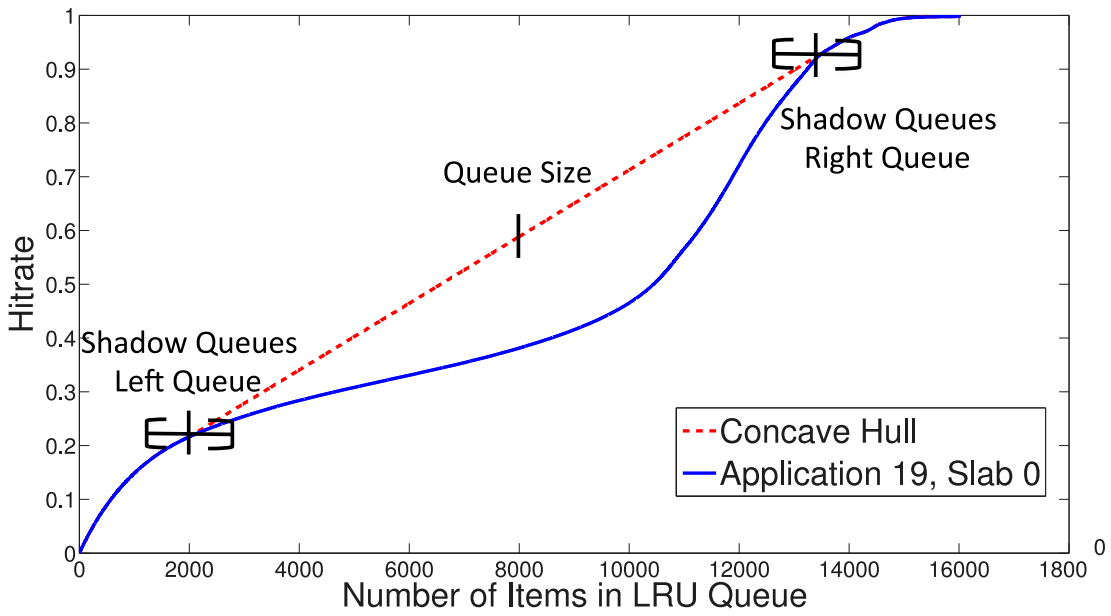

Figure 2.8: Talus example.(taken from [4]).

Talus needs all the hit rate curve to determine if there is a performance cliff, and since Cliffhanger is based on local knowledge, there is no information to construct all the hit rate curve. To overcome this, Cliffhanger determines dynamically if the point is on the performance cliff of the hit rate curve, and it does so by approximate the queue second derivative. If the derivative is positive, the queue is in the performance cliff.

Cliffhanger demonstrates in the test performed [4] that by using its hill climbing algorithm without performance, cliffs can improve the hit rate of a data centre memory cache significantly.

2.6.5 Dynacache: Dynamic Cloud Caching

Memcached allocates its memory to slab allocations greedily based on the sizes of each initial objects at the beginning of the workload. This means that the first requests will determine which slab classes are the pages being allocated. Memcached presents a re-balancing policy to address the above problem. This policy presents low efficiency since a slab only moves when a slab class is the highest requested in thirty seconds, taking a lot of time to adapt to a new requests pattern. We can say then, that Memcached uses a

fixed memory allocation policy and it cannot dynamically adapt efficiently to different application access patterns. A. Cidon et al. [12] said in their paper that one-size-fits-all cache behaviour fits poorly in cloud environments. To tackle this, Dynacache was born.

Dynacache was designed in order to optimally allocate slabs that may be better from new memory allocations. Dynacaches runs as a module that is integrated with the Memcached server. To optimally know to which slab should the memory be allocated, they rely on the follow expressed optimization 2.9. s represents the number of slab classes in Memcached, fi is the frequency of GET operations for each slab class, $hi(mi, e)$ is the hit rate of each slab class as a function depending on the memory (mi) and the eviction policy (e) and M is the amount of memory reserved by the application on the Memcached server. Since the equation depends on the hit rate curve of each slab class, they use stack distances to gather this information. The authors describe the stack distance of a requested item as the length between the top and his current position in the eviction queue. When the item is at the top of the eviction queue the stack distance is 1 and if the item has never been requested the stack distance is infinity. For the equation to be effective, the hit rate curves must be concave or near concave. The concavity can be approximately achieved by using a piecewise-linear fitting. The piecewise-linear hit rate curves allow solving the function using a solver that they call LP Solver. Dynacache, to compute the stack distance, uses a bucket scheme instead of the traditional shadow queue. The bucket scheme is a linked list of buckets, each containing a fixed number of items. The buckets are stored in a queue, and the requests hit the top bucket when the top bucket is full, they remove the bucket at the end of the queue. This way, instead of taking $O(n)$ to compute the stack distance using a shadow queue, it takes $O(B)$ using buckets.

Now that they have the information about how much memory it should be allocated to each slab class, they need to determine which slabs will benefit from this allocations. Dynacache knows which slab classes would benefit by using a metric that they call entropy. Entropy provides a measurement of the uniformity of a distribution function. If the distribution behaves uniformly, the entropy will be high. Likewise, if it behaves deterministically the entropy is zero. By treating the misses of each slab class as probability density functions they can calculate their entropy calling them miss entropy. Dynacache will check the miss entropy to pick the slab classes that are going to be optimized by receiving allocating more memory.

2.6.6 Summary

Memcached allocates its memory in a slab-by-slab way [10]. If Memcached has free available slabs, it will allocate them to the classes that are more requested, originating a problem called slab calcification. Slab calcification occurs when the popularity of requests change and the slab classes with few slabs are now becoming popular. Since the slab classes with few slabs become full quickly, they are forced to evict objects in their class more frequently. It would be beneficial to take slabs that are not being used so

$$\underset{m}{\text{maximize}} \quad \sum_{i=1}^{s} f_i h_i(m_i, e)$$

$$\text{subject to} \quad \sum_{i=1}^{s} m_i \leq M$$

Figure 2.9: Optimal memory allocation equation of Dynacache. (Taken from [12]).

frequently.

One way to address this problem is to manually reset Memcached and let him built its memory over again by filling its cache again. Instead of using the manual approach, rebalancing policies are used. Rebalancing policies auto move slabs dynamically from slab classes depending on their rebalancing algorithm.

In the case of Conglong Li and Alan L. Cox [22], they designed a cost-aware rebalancing policy in an attempt to integrate their GD-Wheel policy [] in Memcached. This design was necessary since Memcached's default rebalance policy was not cost-aware. Every slab class of Memcached knows its average cost per byte information. When an object is evicted from a higher average cost slab class, it will take a slab from the lowest average cost slab class. One disadvantage that may occur is that the lowest average cost slab class will eventually present a small number of slabs.

Another example is Carra's and Michiardi's [10] Cost-based Memory Partitioning and Management in Memcached. It will move slabs from the slab classes that have a lower risk of increasing the number of misses to slab classes that would benefit the most by taking that slab. They know the information of misses by analyzing the miss-rate curve. The miss-rate curve is constructed using two bloom filters. The purpose of these bloom filters is to distinguish the misses from the objects that were not requested before and the objects that were already evicted.

Concerning more generic rebalancing policies, Cliffhanger and Dynacache are two rebalancing policies that can work with any replacement policy. Cliffhanger uses a hill-climbing algorithm in order to know which slab class would benefit from more memory allocation and the classes that would benefit less. The problem of a hill climbing algorithm is that it does not work well when the curve is not concave. To address this, Cliffhanger determines the extreme points where the curve starts and stops being concave. Then it draws an imaginary line between does two points making the curve concave again.

As for Dynacache, it optimally allocates slabs that may profit from new memory allocations. In order to do this, it follows an equation presented in figure 2.9. With this

equation, they have information about how much memory should be allocated to each slab class, needing just to know which classes would benefit from these allocations. To know the classes that would benefit from more memory, they used something they called entropy. Dynacache will check the miss entropy to pick the slab classes that are going to be optimized.

Chapter 3

Proposed Solution

3.1 Introduction

After exploring and analyzing the above sections in chapter 2, we found interesting to explore how can a good memory management that prioritizes more costly key-value pairs in cache, show us a decrease in the total recomputation time of the key-value pairs. If we can decrease the total recomputation time of the values of the key-value pairs even with a higher miss-ratio, it means that applications will still benefit, or benefit more, from this memory management since the time that would be spent on recomputing the key-value pairs for every miss would still be lower. To provide a good cost aware memory management, we invested on two main tracks, a new cost aware replacement policy, and a new rebalancing policy.

To integrate a cost-aware replacement policy in Memcached we needed to have a cost-aware replacement structure capable of preserving more costly key-value pairs in cache, and for this reason, we changed the replacement structure in Memcached. From our research, GD-Wheel [22] may be the best solution available for having a good cost aware replacement structure because it reduces the total recomputation cost and improves the average read latency in Memcached with constant amortized time complexity. To implement such policy in Memcached its necessary to create a different replacement structure, meaning that we have to deal with a multi-threading performance, bigger management between pointers within the structure, and other problems. This structure can lead to possible overheads and concurrency issues, hence due to its complexity, time constraints and the objectives of our thesis, we decided to implement our own replacement policy based on Greedy Dual-Size Algorithm [7], with the aim that we don't change much the original replacement structure of Memcached. Since Greedy Dual-Size Algorithm takes logarithmic time ($O(\log n)$) to sort the priority queue on every insertion, we changed the

replacement structure of Memcached in the attempt to reduce the time complexity.

Our new rebalancing policy was inspired by Carra's and Michiardi's [10] heuristic that moves slabs between slab classes and Conglong Li and Alan L. Cox [22] idea of also having the cost per byte information as a factor in the decision of moving slabs.

In light of what was discussed above, the sections below will describe our proposed solutions. In more detail, in section 3.2 we are going to mention how we planned to change the replacement structure as the replacement policy of Memcached and section 3.3 will mention our approach for a new rebalancing policy.

3.2 Replacement policy

In section 2.5.1, we saw that the replacement structure of Memcached works through recency. The key-value pairs that are more popular and recent tend to have more priority to stay in cache, being the only factor used in Memcached to determine which key-value pairs should be evicted. When important factors like the cost of recomputation of key-value pairs are taken into account, Memcached's replacement structure is not capable of supporting such policies that prioritize the more costly key-value pairs to stay, requiring a re-engineering of the replacement structure to make this possible.

We propose an eviction policy that was inspired by the Greedy Dual algorithm [7]. As described in section 2.5.2 Greedy Dual can be used as a replacement policy that integrates factors like cost and size, it generalizes LRU, balancing items in a priority queue depending on their cost and recency in cache. By using a priority queue, any insertion in cache has a time complexity of ($O(logn)$), because for every item inserted in the priority queue a sort as to be made, to order items in the queue, and for this reason we choose to adapt this algorithm to work in simple queues similar to the ones already used in Memcached. One advantage of doing this, it is because it would be easier to integrate this policy in other key-value stores, without being subject of big overheads in the integration process. Among the queues mentioned above, one of them is reserved for the most costly key-value pairs in the key-value store, and the other queue(s) will function as they usually would in the key-value store with their replacement policy. We preserved the way Greedy dual defines the priority for each item. The reserved queue will have a calculated minimum priority, that determines which key-value pairs should enter and leave the queue.

In the image 3.1 we can see a general and not so specific example of the behaviour of our new algorithm in a key-value store with a LRU replacement policy. When an object p is going to be inserted in cache there are three possible scenarios. The first scenario happens when p is already in memory, and the new priority of p is bigger than the minimum priority of the respective Reserved queue. In this case, p will become the head of the Reserved queue and the tail of this queue, if full, will go to the head of the LRU queue.

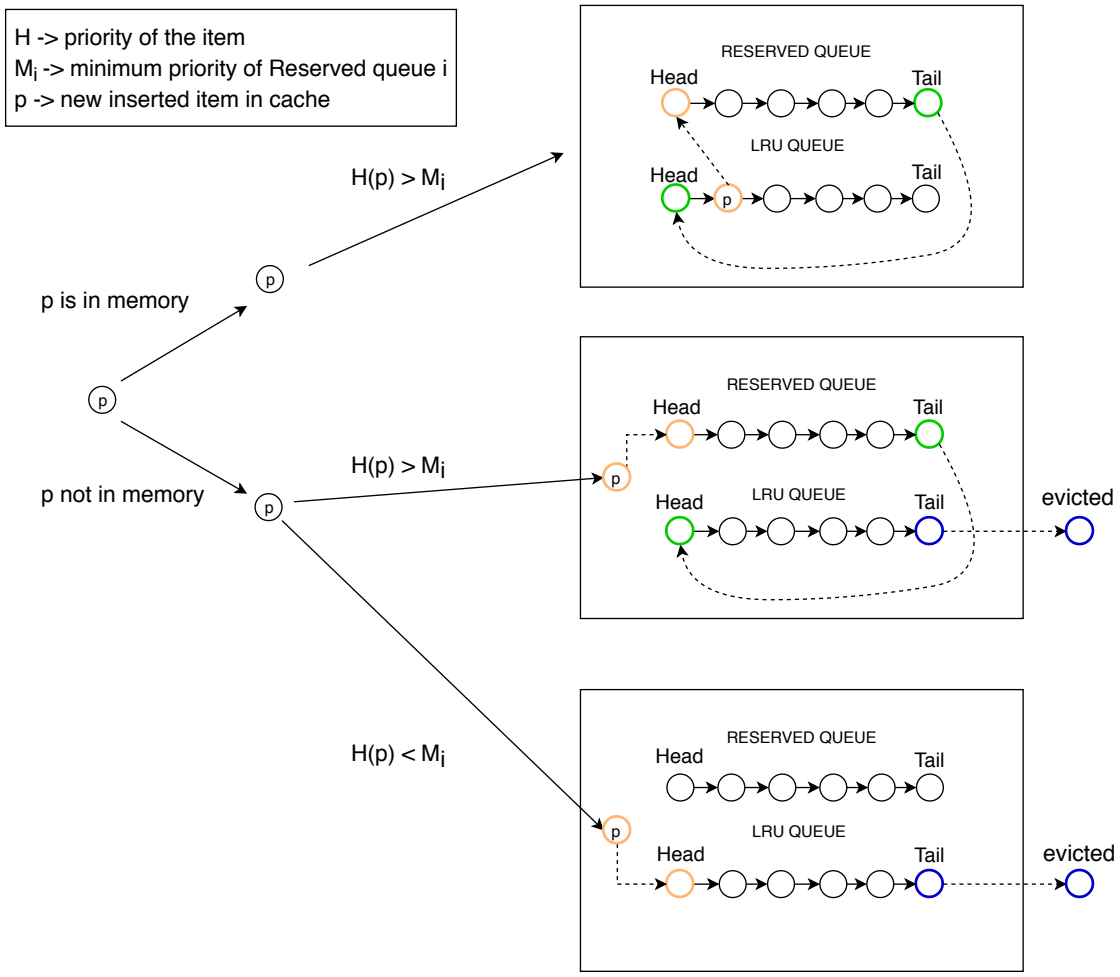

Figure 3.1: New replacement strategy for a LRU based replacement structure key-value store.

The second scenario is when p is not in memory, and its priority is bigger than the minimum priority of the respective reserved queue of p. The tail of the LRU queue, if full, will be evicted, allowing enough memory for p to enter in the key-value store and become the head of the Reserved queue, which consequently causes the tail of this queue to become the new head in LRU queue.

The last scenario occurs when p is not in memory, and its priority is lower than the minimum priority of the respective slab class of p. Evicting the tail of the LRU queue will make room for p to become the new Head of the LRU queue.

In chapter 2.3.1, we saw that Memcached's replacement structure uses LRU queues in each slab class to store the key-value pairs. These queues could be segmented in Hot, Warm and Cold LRU's, leaving recent inserted key-value pairs in the Hot queue, less used key-value pairs in the Cold queue and active key-value pairs in the Warm queue. To combine our strategy with Memcached and preserve its replacement structure, we add one more segmentation that we call reserved LRU. This Segmentation is reserved for storing the most costly key-value pairs in Memcached, leaving the rest of the segmentations of

Memcached to perform the same behaviour as they used to. Since there is a reserved segmentation in every replacement structure of Memcached, we found interesting to change the use of the inflation value in the way the priority of each item is computed. Like we mention before in section 2.5.2, Greedy dual has a global inflation value that is added to each inserted item of its priority queue, and since we have different reserved queues in our proposed strategy it is interesting to have an inflation value for every one of them. With this strategy, the respective reserved queues of the items can gradually increase its priority accordingly to the items' priority in the queue and not in the overall.

For a better understanding, we will designate an item as a key-value pair with the respective metadata. In algorithm 1 we can see our eviction policy behaviour in Memcached. For every inserted item p its priority ($\mathbf{H(p)}$ will eventually be calculated by adding the slab class inflation value ($\mathbf{L}$) and the item's cost ($\mathbf{c(p)}$). Initially, when the memory is empty the items are inserted in the Reserved LRU. When the Reserved LRU is full, the items are inserted in the Hot LRU and the procedure continues the same as in Memcached. After Memcached is full of items, inserting items leads to two possible scenarios, the item is already in memory, or the item is not in memory. When p is already in memory and it is not in the reserved LRU, its priority is updated, and at that moment it will check if the updated priority is higher then the minimum priority of the respective slab class (M_i). If that is the case, the tail of the Reserved LRU moves to the Hot LRU, inserting p to the head of the Reserved LRU and updating the minimum priority if and only if the priority of the removed tail is bigger than the actual minimum priority. Otherwise only the priority of p is updated. When p is not in memory, space will be released from memory by evicting the tail of the Cold LRU, in this instance, L_i is updated if the priority of the evicted item is greater than the actual L_i. Then the priority of p is calculated and compared with M_i, inserting p in the Reserved LRU if it is greater. Otherwise, it will be inserted in the Hot LRU. For p to be inserted in the Reserved LRU it is necessary to make some room, moving the tail of the Reserved LRU to the Hot LRU. In the process of an item leaving the reserved LRU, it is always checked if the priority of that item is greater than the M_i of the slab class, updating the M_i if that is the case.

Our algorithm works around doubly linked lists and it only has to make decisions to know in which segment of the LRU the items are going to be inserted. Unlike the Greedy Dual, it provides a time complexity of $O(1)$ on insertions. In order to obtain such time complexity, we discard the possibility of sorting the reserved queue when an item is inserted. Instead, we restrain part of the memory of the key-value store and **eventually** the most costly items will be gathered in the reserved queue in Memcached. When inserting in the reserved queue, we know that the items may have a big potential to be in cache. At the same time, we know that when removing an item it doesn't matter if it is not the less costly item in that queue because we believe that removing the less or bigger item in the collection it will not have a big impact in the end. Also, the variable minimum priority will restrain the items which will enter in this queue, leading to allowing only the most costly items to enter.

Algorithm 1 Eviction of objects in LRU based on cost priority

```
1: For each slab class i
2: Initialize M_i ← ∅ and L_i ← 0
3: function OptimizedGreedyDual
4:     For each requested object p in slab i
5:     if p is already in memory then
6:         H(p) ← L_i + c(p)
7:         if p ∉ RESERVED_LRU and RESERVED_LRU is full and H(p) > M_i then
8:             t ← RESERVED_LRU.pop()
9:             HOT_LRU.put(t)
10:            if H(t) > M_i then
11:                M_i ← H(t)
12:            end if
13:            RESERVED_LRU.put(p)
14:        end if
15:    end if
16:    if p is not in memory then
17:        if Not enought memory to store p then
18:            Evict q from tail of COLD_LRU
19:            if L < H(q) then
20:                Let L_i ← H(q)
21:            end if
22:            H(p) ← L_i + c(p)
23:            if H(p) > M_i then
24:                o ← RESERVED_LRU.pop()
25:                RESERVED_LRU.put(p)
26:                if H(o) > M_i then
27:                    M_i ← H(o)
28:                end if
29:                HOT_LRU.put(o)
30:            else
31:                HOT_LRU.put(p)
32:            end if
33:        else
34:            H(p) ← L_i + c(p)
35:            if H(p) > M_i or RESERVED_LRU is ∅ then
36:                RESERVED_LRU.put(p)
37:            else
38:                HOT_LRU.put(p)
39:            end if
40:        end if
41:    end if
42: end function
```

3.3 Rebalancing policy

Slab classes allocate their memory accordingly to the distribution of items size. When Memcached allocates all the possible memory some sizes were requested more times than others. Those respective slab classes that are more requested allocate more memory than the others to be able to store the more requested items. If there is a change in the distribution of the items, the hit ratio suffers a decline due to the fact that more memory was allocated to the former popular items. The new popular items do not have enough space in memory to build their popularity. Reallocating some memory to new popular slab classes may increase the hit rate since it would no longer be necessary to evict some items because there would be more space for the new popular ones to be stored.

It is important to increase the hit rate, in other to provide to the client faster results. More misses means that Memcached needs more recomputations of the requested items, resulting in a longer time to return a result to the client. Some items take more time than others to recompute, and we believe that by focusing more costly items to stay in cache it can lower the overall recomputation time. One way to do this is by reallocating slabs taking into consideration the cost of items' recomputation.

Memcached's original rebalancing policy does not take into consideration the cost recomputing of one item when determining the appropriate slab class for allocating new memory, and therefore, a new rebalancing policy is needed. The proposed rebalancing policy in algorithm 2 was inspired by Carra's heuristic together with GD-Wheels rebalancing policy. Both are explained in section 2.6.3 and section 2.6.2 respectively. Our policy reacts when there is a slope in the hit ratio curve following by the movement of slabs from the slab class which has the lowest miss ratio and average cost per byte, to the slab with the highest miss ratio and average cost per byte. This means that the policy focus on taking slabs from the less popular class, which has the lowest average cost per byte and gives them to the most popular slab class with the highest average cost per byte. We do that because, we think that reallocating slabs to the popular slab class with the highest cost items it will open more opportunities for new higher cost items to stay in cache, lowering the total cost of re-computation of the items.

Algorithm 2 Eviction of objects in LRU based on cost priority

1: **function** SLOPEANALYSER
2: **For every slope in the hit-rate**
3: NewRebalancingAlgorithm
4: **end function**
5:
6: **function** NEWREBALANCINGALGORITHM
7: $m_i \leftarrow$ Misses from slab class i
8: $cb_i \leftarrow$ Avarage cost per byte from slab class i
9: $r_i \leftarrow$ Requests from slab class i
10:
11: $id_{\text{take}} \leftarrow argmin(\frac{m_i cb_i}{r_i})$
12: $id_{\text{give}} \leftarrow argmax(\frac{m_i cb_i}{r_i})$
13: **end function**

Chapter 4

Implementation

4.1 Memcached Components

The policies proposed in section 3 were implemented from the stable released version of memcached-1.5.8. Memcached is implemented in C and is composed by a set of modules, each one has its purpose and functionalities. The modules that we tackle are the following:

- **memcached.c.** The connections with the clients and the Memcached protocol are implemented in this module.
- **item.c.** Here is where it is done the management of the memory of the client's items.
- **slabs.c.** The management of the memory and the slabs of Memcached are carried out on this module.
- **threads.c.** In this module is where the threads are managed. For instance, every operation that involves initialising or triggering events that will result on how threads function, are done in this module.

4.2 Memcached protocol and metadata

For storing requests, Memcached offers a list of API's such as Set(Key,Value), Add(Key,Value) and Replace(Key,Value). In order for Memcached to obtain the information about the cost of each item, it is required an extension of these commands. The new commands have the cost of the items' recomputation that will be stored, and this information comes as an extra argument, i.e. Set(Key,Value,Cost), Add(Key,Value,Cost) and Replace(Key,Value,Cost). We adopted to use Memcached's ASCII protocol for communicating with the clients, and since we add an extra argument to some commands, we

needed to adapt the protocol to cover this new variable. To do so, we change the function process_update_command() to read an extra 32-byte int from the command line. The command line as the format of: [command name]|[key]|[flags]|[exptime]|[bytes].

- command name: Can be add, set, replace, prepend, append.
- key: The key associated with the data to be stored
- flags: Arbitrary 16-bit unsigned integer that the server stores and sends back when the item is retrieved. The client can use this flag to store specific data information.
- exptime: It is the expiration time. If the exptime is 0 then the item never expires. Otherwise, the item cannot be retrieved by the clients if the item stays longer than the exptime.
- bytes: Represents the number of bytes to follow in the data block.

The changes done in the process_update_command() have resulted in the function to expect a command line with the format of: [command name]|[key]|[cost]|[flags]|[exptime]|[bytes].

Focusing on the changed metadata, we first added float in every slab class to represent the inflation value and use it to calculate the items' priority. To enable our replacement and rebalancing policies, was necessary for each item to store its cost and priority. We increased the item's metadata by 4 bytes, to be able to store this information. Taking into consideration what Conglong li said [22], the size of allocating the metadata is rounded up to 8-byte boundary to avoid fragmentation, it was confirmed that the extra 4 bytes would not have any impact on the allocated size.

4.3 Eviction policy

4.3.1 Changes of the LRU structure

Regarding the LRU structure of Memcached, in order to implement in every slab class the mentioned reserved area that contains the most requested costly items, it was necessary to change the actual structure of the LRU. Therefore we will first explain in this section how the replacement structure is implemented in Memcached. Then we will explain the changes to the original replacement structure. As previously explained in the section 2.5.1 that Memcached's replacement structure is composed by a group of doubly linked lists, distributed to each slab class. These queues can be segmented in three layers: Hot, Warm, Cold. There is a fourth layer called Temp, which is only used when a flag is activated. The flag receives a number, that will define the temporary time to live of the items (TTL). The LRU crawler that scans throw all slab classes will make items with their TTL greater or equal to the temporary TTL to go to the Temp segment. Items stored in Temp are never bumped within its LRU or moved to other LRU's. They also cannot

be evicted. The Temp segment helps reduce holes and load on the scanning of the LRU crawler. In figure 4.1 we can see an example of how Memcached manage its segments in a LRU. There are two distinct arrays, one for storing the heads of the lists and the other for storing the tails. To better clarify the function of these queues, we are going to focus on explaining how the array of heads work, but keeping aware that both the array of heads and the array of tails work in the same way.

To better manage the segmentations in Memcached, the array of heads keeps track of the heads of every segmentation in every doubly linked list in a strategic index which will be described below. There are a total of 256 indexes, in which the range of index 0 to 63 stores the heads of the Hot segmentation of the LRU, the index 64 to 123 belongs to the heads of the warm segmentation, from 124 to 191 refers to the heads of the cold segmentation and from 192 to 255 are stored the heads of the temp segmentation. As an example, if an item wants to insert in slab class 7 in the hot segmentation, it will be the new head in index 7. Now instead of the item being inserted in the hot segmentation, let's imagine that we want to insert the item in the warm segmentation, this means that the item will be the new head in index 71 (7+64) and if it were inserted in the cold segmentation it would be stored in index 131 (7+124).

Since the Temp segment is rarely used, and it is only activated if the client wants to, we used space in the replacement structure of the Temp segment for our reserved area. Hence each slab class has a reserved area and the capacity for storing the items, will be determined by number passed in the flag used to activate the Temp LRU. Like we mentioned, the flag receives a number and in essence, this number represents the total number of items that the reserved area can contain in each slab class.

4.3.2 Priority of the items

As described in section 3.2, a priority is needed to determine if an item is worthy or not to enter its respective reserved area. This way we chose to use the priority referred in algorithm 1 ($H(p) = L_i + c(p)$) as the decision factor. Two functions were created *return_it_priority(int id, const unsigned short cost)* and *it_new_priority(item *it, const unsigned short cost, int id)*, returning a simulation of the priority that the item could have, and respectively the other function determines and stores the priority in the metadata of the item. The id in both functions refers to which slab class the item will be stored, being necessary to know since the inflation value (L_i) is needed to determine the priority of the item(p). The reason for having a specific function that simulates the priority of the items is because when there is no more memory, an item needs to be evicted from the cold segmentation. If the new item has enough priority to be in the reserved area, an item from the reserved area needs to make room and jump to another segment.

The priority of the item is calculated every time it is inserted in Memcached, and the priority is recalculated on a GET hit. By recalculating the priority of the items on every GET hit, we give the possibility for popular items to build their priority and compete

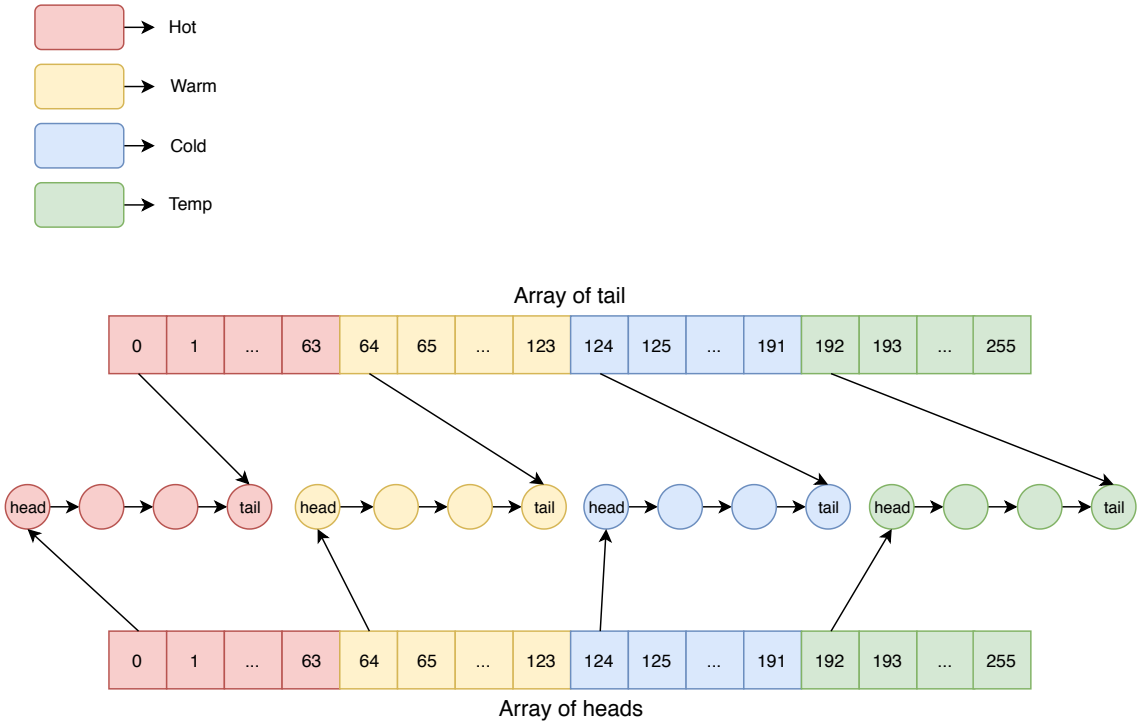

Figure 4.1: Replacement structure of memcached.

with other higher priority items.

4.3.3 Eviction and Update of items

Recapping the process of how an item is managed after a storage command in Memcached, there are two possibilities, the item is in memory, or it is not. Independently if the item is or is not in memory, we saw that before the item is placed or updated in the LRU structure it is first stored in memory, and for this reason, Memcached first checks if the memory is full, evicting an item and leaving room for a new one if necessary. The first part of this process is handled by the function *do_item_alloc*(), which checks with the function *do_item_alloc_pull*() if an eviction is required before the item is stored in memory. After the function *do_item_alloc_pull*() the *do_item_alloc*() ends the function by initializing the new item with the respective metadata information. The second part that involves storing the new item in its respective replacement structure is controlled by the function *do_store_item*(). In this function, is decided every administration of the new items in the replacement structure such as conducting the items between the segmentations in the LRU and deciding where the item is stored. At the beginning of the function, *do_store_item*(), it starts by first searching the key of the new item to see if it already existed in replacement structure. If an item with the same key is found, it is replaced by the new item that was recently inserted in the memory. That means that the old item that was replaced is released from memory and the LRU structure, and the new item with the updated metadata is inserted in the LRU. If the item is not already in the

memory, it is simply inserted in the LRU structure.

To change the way items are managed in Memcached through their priority instead of their recency, we needed to change the functions mentioned above to modify this process:

do_item_alloc(). An extra argument of the cost of the item was added to calculate the priority of the item and to fill this information in the metadata. At the beginning of the function, the priority of the item is calculated and is passed in the function do_item_alloc_pull(), which was also modified and was added an extra argument. The reason for this decision was because of the introduction of the reserved area in Memcached. Now that items can be stored in the reserved area, when this area is full and the new items have greater priority than the minimum priority of the respective slab class, this new items will be inserted in the reserved area, forcing older items in the reserved area to move to the hot segment of the LRU to make room. Otherwise, if the above conditions are not met, the newly inserted items are placed in the hot segment of the LRU. For the mentioned reasons, we needed to know the priority to know when to handle an eviction with a potential item to enter in the reserved area. Another change done in this function was the manipulation of the items id. If the item has potential to be in the reserved area we need to assign the correct id as we mention in section 4.3.1.

do_item_alloc_pull(). In this function, memory is allocated to newly inserted items and if memory is needed, evictions of items are made in the cold segmentation. We added a new argument to this function with the priority of the newly inserted item, because depending on the priority, a different process of eviction occurs. The function *lru_pull_tail*() is responsible for handling this evictions. If the item as enough priority to be in the reserved area we tell the function *lru_pull_tail*() to evict an item from the cold segment of the LRU respective its slab class and move the tail of reserved area to the hot segmentation of the LRU, leaving this way an empty space for inserting the new item. Otherwise, an eviction of an item of the cold segment is done and the new item does not belong to the reserved area.

lru_pull_tail(). The periodic maintenance of the LRUs happens here. It has the possibility of doing evictions, check if items are expired or simply manage the items from the LRU, by swapping items over segments. This function can receive the id of the slab class of an item, the segmentation that the item belongs to, some flags that indicate if an item is going to be removed or swapped to another segment or simply nothing to check the expiration of the item, the total bytes that the total removed items from the hot an warm segment can achieved, the max age of a certain item in the warm segment of the LRU can achieve, and a reference to an potential item that was removed if we want the function to return it. To know what should happen to an item there is a switch that receives the information of the what segmentation that the item belongs to, and depending on the flags it will swap between segments or update the item's metadata. Having into account that a new segmentation was needed to contain the reserved item, we added a new layer to the switch. This new layer provides the behaviour that the new segmentation will have upon receiving a new potential item. When we indicate the function that an item

is from the new reserved segmentation, it will first start by locking the LRU to perform a search on the cold segmentation to see if there are items to evict. If there are no items in the cold segmentation, it will try to find items for eviction in the other segmentations by order of: warm segmentation, hot and then in the reserved segmentation. The lock was done to ensure that multiple threads do not alter the LRU, and eventually creating concurrency problems. After the eviction of the item, the inflation value is updated with its own priority if the value is greater than the inflation value. Now, after some memory is released, we will swap the tail from the reserved segmentation to the hot segmentation. Since the item had a good potential to be in the reserved area, we thought that by putting it in the hot segmentation it would bring more opportunities to be more requested becoming a better candidate to enter the reserved segmentation again. With the swapping done, if the priority of the swapped item is greater than the minimum priority of the reserved segmentation, it is updated, and after, we unlock the LRU since we are not doing more changes.

do_item_update(). Every time an item gets a hit, or it is updated, this function is called to take care of this process, meaning that it will manage the transition between the LRU segments. Upon an item receiving more than one GET hit, the item becomes active. When this happens, the item is then bumped to the warm segment. Otherwise, if the item receives just one hit, it will be pushed to the head of its own segment to avoid to be easily evicted.

We extended this function by adding one more condition for when items are updated. The condition says that when an item is updated, if its priority is bigger than the minimum priority of the reserved segmentation, the item will be bumped to the reserved segmentation, bringing the reserved's tail to the hot segment to make space for the new inserted item.

4.4 Rebalancing Policy

In order to get all the information required, to apply our proposed heuristic mentioned in section 3.3 we gathered the information as follow:

- **Misses from each slab class.** Memcached as information about the total misses that happen in its own system, but it does not provide us with the information on the misses for each slab class. In the scenario that every miss is followed by the storage (SET) of that item by the client, we address the problem above by considering an insertion of that item as a miss. After the warm up phase, we send a signal to Memcached and to count every insertion as a miss for that slab class.

- **Requests from each slab class.** To obtain the requests done to each slab class, we add the misses and hits of each one. We described above how do we get the information about the misses. Memcached already stores the information about the hits in

each slab class for static reasons. We can benefit from this and use this information to obtain the number of requests in each slab class.

- **Average cost per byte from each slab class.** We created a new float variable in each slab class to store the information about their respective total cost of items they store. For every insertion of an item in the LRU, we add its cost in the variable that we mentioned. The same thing happens when we remove an item, we decrement the variable with the cost of the removed item. To obtain the avarage cost per byte of a slab class, we created a function that returns this information by dividing the total cost of that slab class with the size of that slab class.

Our rebalance policy reacts when there is a drop in the hit ratio curve. We define this procedure by starting to allocate memory for storing the information of the hit rate per each second. We store this information in an int array of 200 positions. For every second, a position is filled in the array starting from the index 0 to the index 199. When the pointer reaches the end of the array, it starts to fill from the beginning on the index 0 overwriting the information that was there and overwriting the following positions as the seconds' pass. To gather the information about the hit rate and analyse it, we created two event handlers in the main loop of Memcached that we are going to explain below. Memcached uses libevent [23] to provide callback functions when an event occurs on a file descriptor or after a timeout. Some examples of events that are used are the management the socket connections between clients, to update Memcached's clock, to control the LRU maintainer, and many others. The main loop event that we mentioned earlier is the event that occurs in each second to update the current time of Memcached. We took advantage of this mechanism and after the time is updated we call the information handler that gathers the information about the hit rate, and after four times that the information handler is called, an analysing handler is triggered. The analysing handler will check the next four index from the last position that it stopped, i.e. it starts checking from position 0 to 3 and the next time it will check from position 3 to 6. In every index that the analysing handler goes through, it will see the distance between the maximum point that was seen, and it will compare if that distance is bigger than 10% of the maximum point. If this condition occurs, we assume that the hit-ratio as dropped significantly enough and we will force it to do a rebalance of the memory. We tell Memcached to give one slab from the slab class that as the higher $\frac{m*cb}{r}$ to the slab class that as the lower $\frac{m*cb}{r}$, where m is the misses, cb the average cost per byte and r the requests, all from the respective slab class.

Chapter 5

Evaluation and Analysis

5.1 Introduction

This chapter reports our evaluation work and is divided into two parts, the methodology and the results. Our methodology was inspired by the work of Conglong li's [22], with the purpose of a comparing that work with our own in the future. Taking this into consideration, our experimental environment and workloads are similar to theirs so that a better comparison can be made. After the methodology is presented, we report our main evaluation results. Our goals are to analyse the behaviour of our replacement and rebalancing policies. To show the efficiency of our replacement policy, we use workloads where all data objects have the same size. By doing this, we only use one slab class of Memcached, not letting the rebalancing policy interfere with the results. To complement these experiments, we also use workloads that manipulate objects with varying sizes, to exercise the use of different slab classes. This way, the rebalancing policy can operate across this slab classes.

5.2 Experimental Environment

Our experimental environment consists of two machines both equipped with an Intel Xeon E5-2620 v2 (with Hyper-Threading processor), 64 GB of RAM and 2 NIC Intel Corporation I210 Gigabit Network Interfaces. One machine was used to execute the Memcached and the other hosts the execution of two YCSB versions, the modified YCSB with the modified spymemcached and another with the normal spymemcached. Our experiments will compare our own version of Memcached with the original implementation serving as baseline. The reserved segmentation of our algorithm will have capacity to hold 10% of the items in the slab class of all the experiments. The normal YCSB will

execute workloads on the original Memcached, and the modified YCSB will execute workloads on our modified Memcached. We configured Memcached to use 8 threads with 2GB and 10GB of RAM. We will In the presentation of our results, we use LRU, to denote the original replacement policy of Memcached and OUR-LRU, to identify our proposed replacement policy.

5.3 YCSB benchmark

The Yahoo! Cloud Serving Benchmark (YCSB) is an open source load generating tool that was designed for evaluating and performance comparing of distributed NoSQL key-value stores. The benchmark consists on two phases, the loading phase, which loads the key-value store with SET operations and the measurement phase that executes and evaluates a set of operations to the key-value store depending on the desired workload. A workload is defined by a test scenario with certain features like the number of transaction to the key-value store, percentage of GET, SET, UPDATE operations, and others. This benchmark offers an extensible workload generator creating the opportunity to add additional workloads that can measure different scenarios for key-value stores, making also easy for the client to adapt to the benchmarks new data serving system. For these reasons, we choose to use this benchmark. In each loading phase, we warm up Memcached until it has a controlled hit rate of 95%, using the same setup to our proposed replacement policy for a fair comparison. In the measurement phase, we send 100 million GET request to Memcached with a *zipfian* distribution on the keys. For every miss on the GET request, we send a SET request to Memcached. If the hit rate stays on the 95% we expect to have about 5 million SET request for 100 million GET requests. We do the same procedure to the Memcache running our proposed replacement policy. We repeat each workload on YCSB at least three times to report the average on the results obtained in each independent run.

5.3.1 Changes to YCSB benchmark and the Client

To create an experimental environment that could handle an evaluation of the benefits of storing more costly items in Memcached, we change the core workload of the YCSB and the Memcached client. In the core workload of the YCSB, we add a new feature that allows choosing the percentage of three types of costs, the low, medium and high. Each type is picked based on a uniform distribution, meaning that each type will be called as many times as the percentage that was assigned to them allows. The low type generates a cost ranged from 10 to 30, the medium from 120 to 180 and the high from 350 to 450. The cost for each type of object is selected for the corresponding cost range described above and follows a distribution from another feature that we added. This cost distribution can be *uniform* or *zipfian*, but we will use an *zipfian* distribution in our experiments. YCSB uses spymemcached [33] as the client to interact with Memcached. We changed the ASCII protocol of spymemcached to add the cost of the key-value pairs,

Workload	Cost Distribution	Key/Value size (Bytes)
BASELINE	10-30(80%);120-180(15%);350-450(5%)	16 / 256
RUBIS	10-30(20%);120-180(75%);350-450(5%)	16 / 256
TPCW	10-30(50%);120-180(25%);350-450(25%)	16 / 256

Table 5.1: Single value workloads (inspired in [22]).

Workload	Cost Distribution	Key/Value size (Bytes)
BASELINE	10-30(80%);120-180(15%);350-450(5%)	16 / (192/256/320)
RUBIS	10-30(20%);120-180(75%);350-450(5%)	16 / (192/256/320)
TPCW	10-30(50%);120-180(25%);350-450(25%)	16 / (192/256/320)

Table 5.2: Multiple value workloads (inspired in [22]).

using the costs generated from YCSB. This way we can connect with Memcached and send him the information about the cost of each key-value pair.

5.4 Workloads

5.4.1 Single size workloads

In Table 5.1, we present the representation of our workloads inspired on Conglong li's workloads [22]. The criteria to choose these workloads was due to the fact that they were based on parameters extracted from a real web application, that are denoted as RUBIS and TPC-W. We start with workload 1, which is our baseline having three cost proportions with Zipfian distribution and values of 256 bytes. Workloads 2 and 3 represent the workloads of RUBIS and TPC-W with the cost proportions of the two benchmarks, respectively.

5.4.2 Multiple size workloads

In Table 5.2, we present the workloads that use multiple size objects. The workloads were also inspired by on Conglong li's multiple size workloads [22]. The cost proportions of these workloads are the same as in Table 5.1. The higher the cost, the higher the value of the key-value pair, e.g., if the key-value pair has a cost of 360 the size of the value is 320 bytes and if the cost has between 120 and 180, in this case, the value would have been 256 bytes.

5.5 Results

5.5.1 Single value Workloads

Average GET/SET latency We represented in Figure 5.1 the average GET Latency for the baseline workload with the different defined cache sizes from our experimental environment. The average Get Latency is around 222 μs within different replacement policies

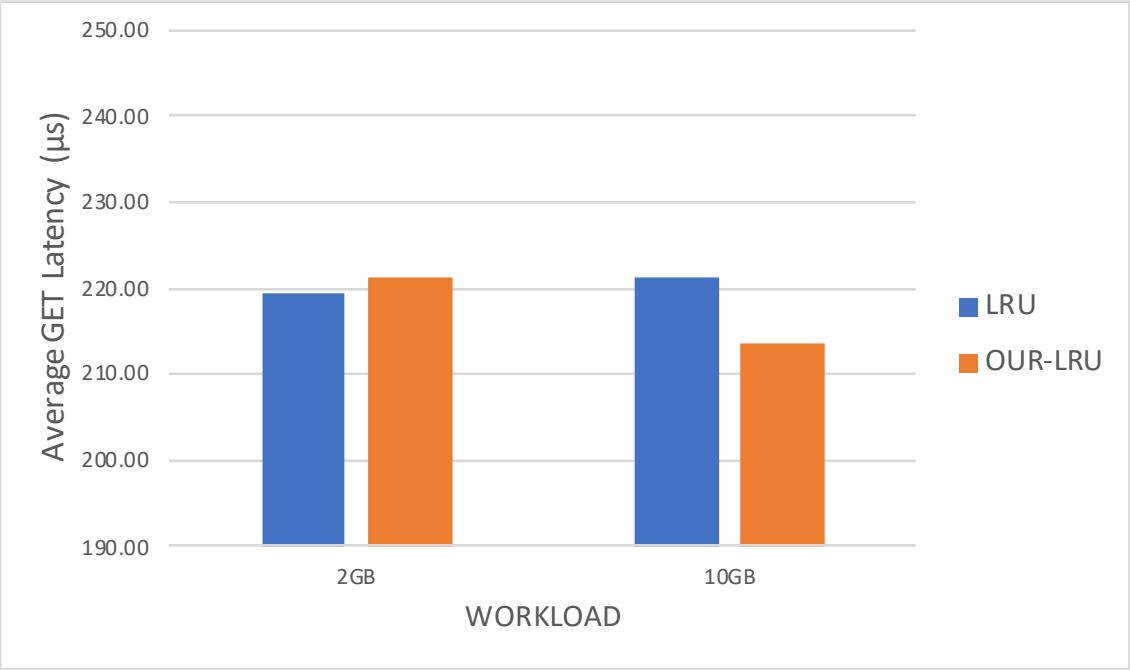

Figure 5.1: Average Get Latency (μs) for the baseline single workload for different cache sizes.

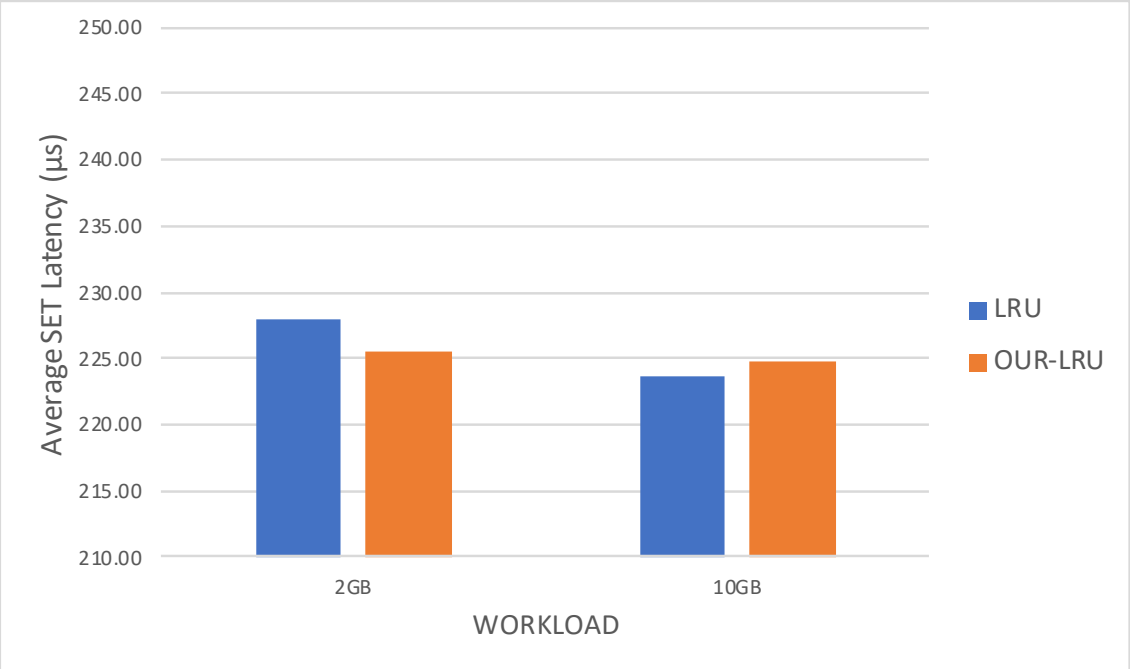

Figure 5.2: Average SET Latency (μs) for the baseline single workload for different cache sizes.

Workload	LRU	OUR-LRU
BASELINE	95,027%	95,0293%
RUBIS	95,0292%	95,0287%
TPCW	95,0306%	95,0288%

Table 5.3: GET Hit rate for each single workload in Memcached 10GB

in different Memcached sizes. When Memcached receives a GET request, there will be a lookup in the hash table to know if the item exists in memory, updating its priority and bumping to other segments in the replacement policy if the items exist and have the conditions to jump between segments. After this process, the response of the GET request is sent to the user, leaving us to conclude that the complexity of our proposed solution does not increase in the GET requests. Figure 5.2 reports the average SET latency observed for the baseline workload for the different Memcached sizes. The average SET latency of our LRU is 225 μs which is close to the original LRU of Memcached. This show that our proposed solution does not raise the complexity on the SET operation. A possible explanation for this results is that since we do not order our reserved segment we only use insert and remove operations between doubly linked lists, resulting in constant time complexity of $O(1)$.

GET Hit-Rate The results for both cache sizes show that they have very similar behaviour. Hence we only show in table 5.3 the hit rate for the GET request for the Memcached server of 10GB RAM. For the single workloads, our LRU shows a variation in the hit rate between the original LRU of 0,00153%. Since the number of misses does not increase, the number of recomputation times of the objects does not also increase, reducing the total recomputation cost of objects.

Average Access Latency Since in our Experimental Environment, we did not use a database to recompute the key-value pair, we will describe how we calculate the access latency. For every GET request, we mentioned that it takes an average of 222 μs to handle the operation, we consider this value to be the latency for every successful GET request. For every miss, we have an extra delay in the response of the GET request due to the recomputation of the key-value pair. When a miss occurs for the lowest recomputation cost of 10 we assume it is twice the latency of a GET hit (444 μs), leading to a 44,4 μs for each cost unit.

In figure 5.3 and 5.4 shows the average access latency of each single workload for Memcache of 10 GB and 2 GB respectively. From seeing the figures, we can understand that our LRU for a Memcached server of 10GB it will not benefit much since the values of the average access latency are close to the values of the original LRU of Memcached. On the other hand, for a Memcached server of 2GB, we can see that our LRU reduces the average access latency of 6%. For the workload RUBIS it can decrease the average access

latency up to 7%.

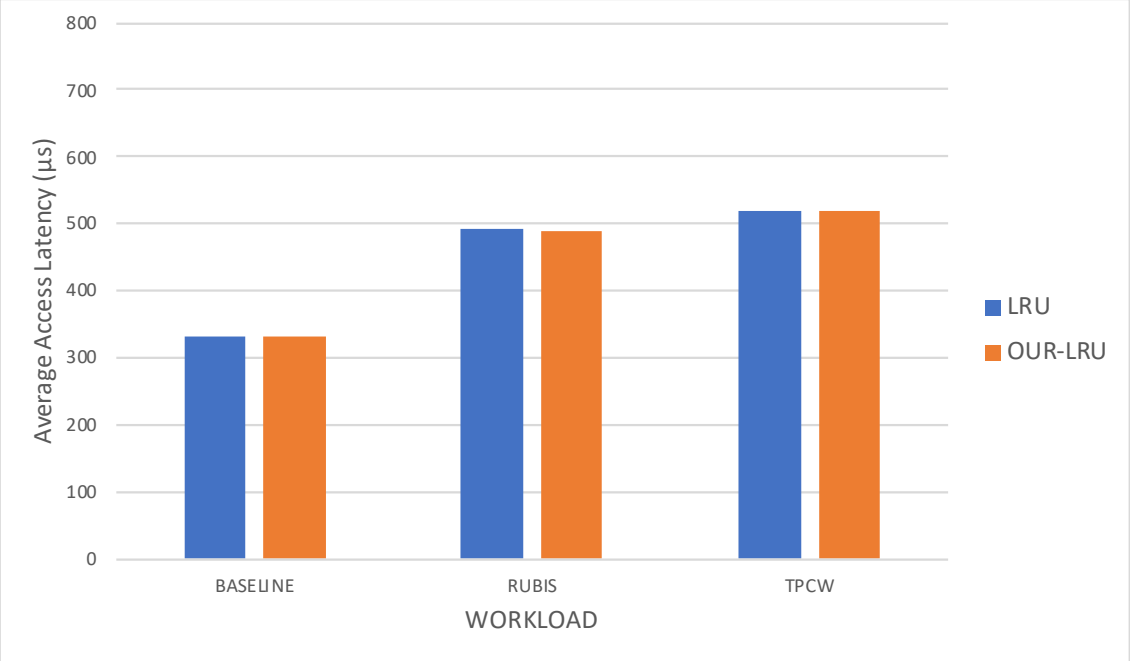

Figure 5.3: Average Access Latency (μs) for the single workloads in Memcached 10GB RAM.

Total Recomputation Cost Taking into account that the hit rate between the original LRU and our LRU is practically the same, the reason for the variation of average access latency between the replacement policy resides in the total recomputation cost. In figures 5.5 and 5.6 we can see the normalized total recomputation cost in Memcached server of 10GB and 2GB of RAM, respectively. To do the normalization, we set the results of each workload for the original LRU to be 100% and results our LRU are normalized based on the original LRU. As one could except, since the average access latency did not change significantly in the Memcached server of 10GB, it is normal that the total recomputation cost would be the same as the original LRU. In the total recomputation cost of the Memcached server of 2GB there is a decrease in total recomputation cost of up to 7%.

5.5.2 Multiple values workloads

Average Access Latency In the LRU and OUR-LRU policies, there was no rebalancing policy, meaning that no slabs were not moved in Memcached. The LRU-REBAL and the OUR-LRU-OUR-REBAl, there was a rebalancing policy running together with the replacement policies. The LRU-REBAL is the original LRU with the original rebalancing policy of Memcached, and the OUR-LRU-OUR-REBAL is our replacement policy with our proposed rebalancing policy. In figures 5.7 and 5.8 it is represented the average

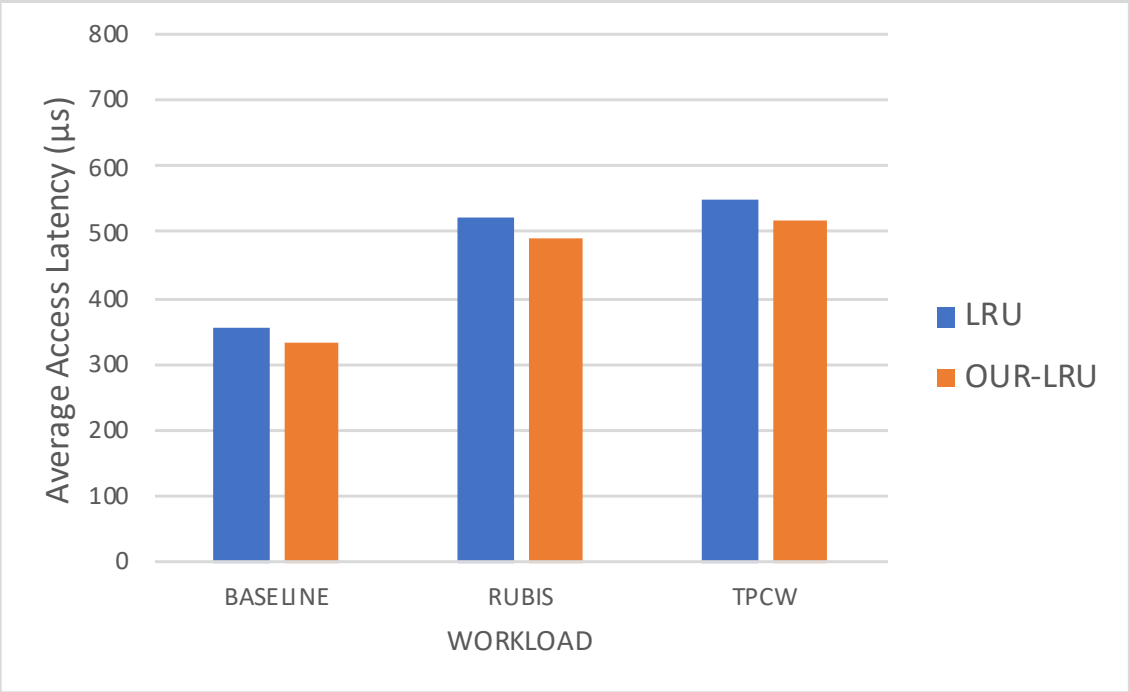

Figure 5.4: Average Access Latency (μs) for the single workloads in Memcached 2GB RAM.

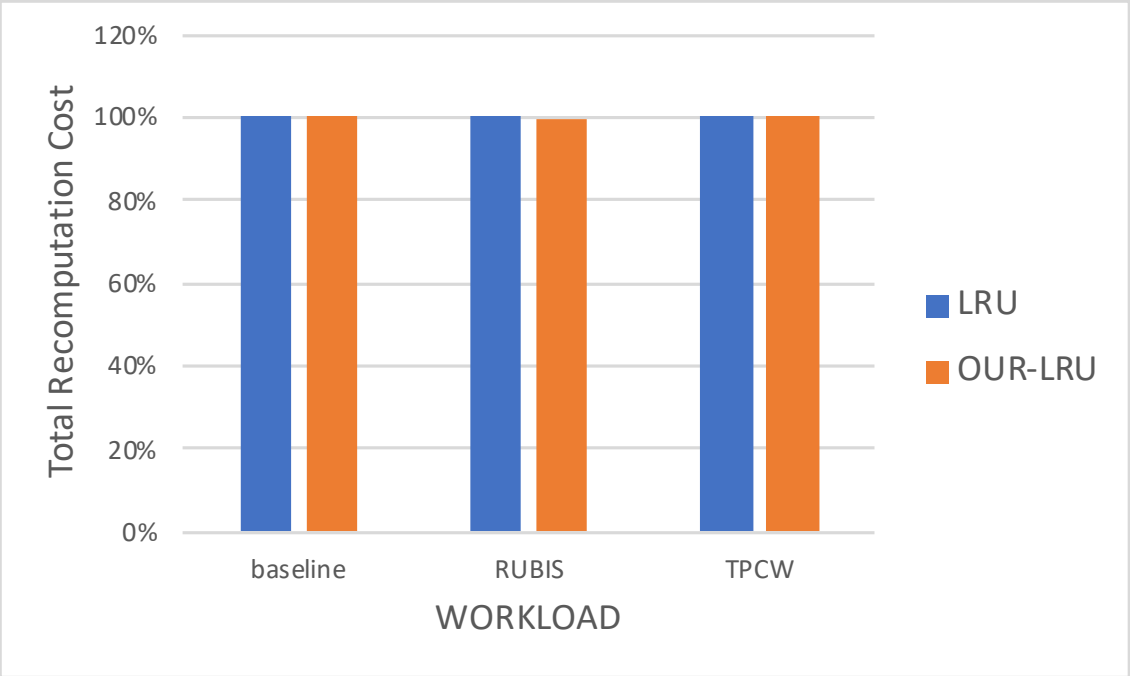

Figure 5.5: Normalized Total Recomputation Cost (μs) for the single workloads in Memcached 10GB RAM.

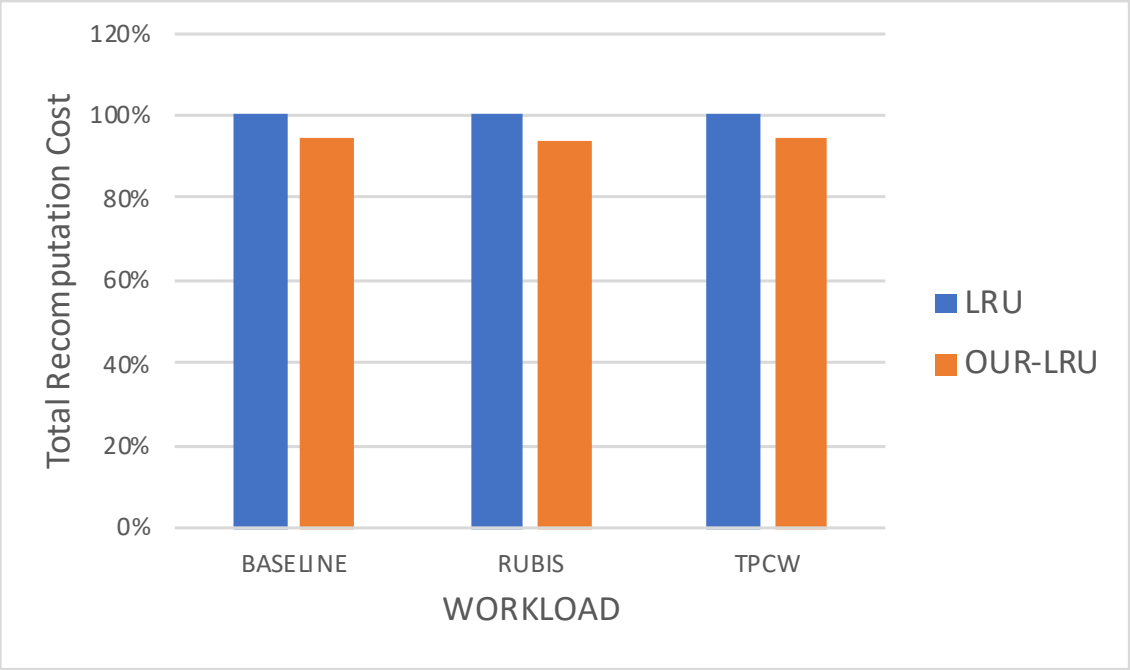

Figure 5.6: Normalized Total Recomputation Cost (μs) for the single workloads in Memcached 2GB RAM.

access latency of the multiple value workloads on a Memcached with 10GB and 2GB of RAM, respectively. We applied the workloads in four possible scenarios that were described above. We calculate the average latency the same way we did in the single value workloads.

In Memcached server with 10GB of RAM, we notice that there is no improvement with our LRU since both our LRU and the original LRU present the same average access latency. On the other hand, there was an improvement in the average access latency between the original replacement and rebalance policy and our replacement policy and our own rebalancing policy, which resulted in a decrease of about of 3%.

Now if we look to the results on the Memcached server 2GB of RAM, we see improvement both with our replacement policy and or rebalancing policy. With our LRU we can see an improvement up to 7% comparing to the original LRU, and we can also see an improvement up to 8% with or rebalancing policy comparing to the original rebalancing policy of Memcached.

Total Recomputation Cost The normalization was done the same way as we did in the single value workloads total recomputation cost, with the difference that we now put also the LRU-REBAL as 100% to serve as a reference to deduct the value of our LRU with our rebalancing policy. In Figure 5.9 we can notice that with our rebalancing policy, we can improve the total recomputation cost in 2,5% up to 3% with the RUBIS workload. As for Memcached server with 2GB of RAM, Figure 5.10 reports results that show that was a slightly bigger decrease with our rebalancing policy. There was an average improvement of almost 6% with our rebalancing policy and an improvement up to almost 8% in the

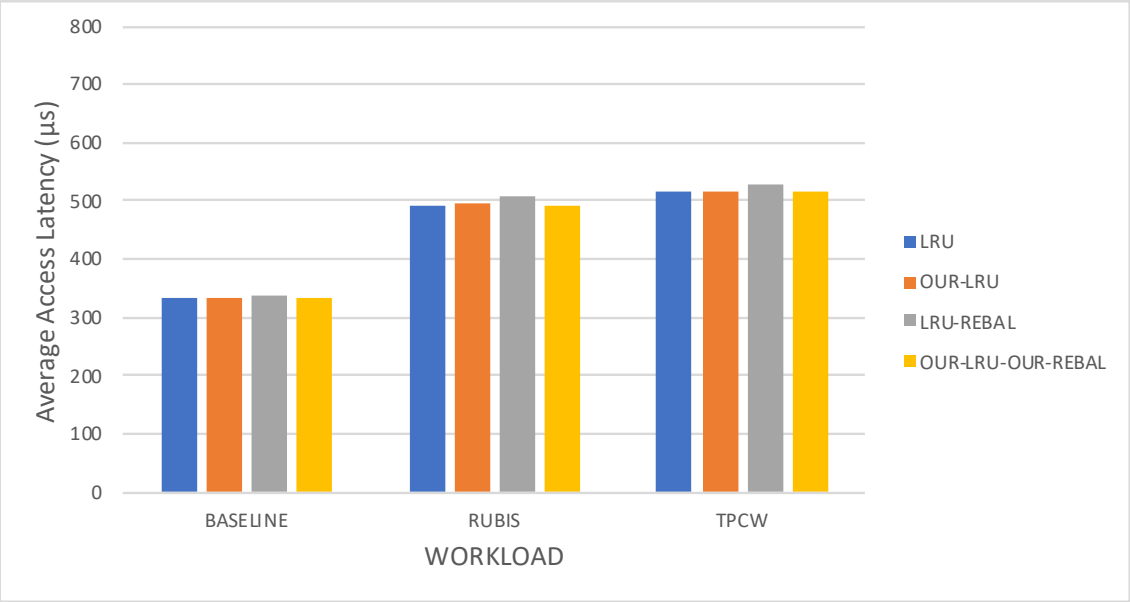

Figure 5.7: Average Access Latency (μs) for the multiple value workloads in Memcached 10GB RAM.

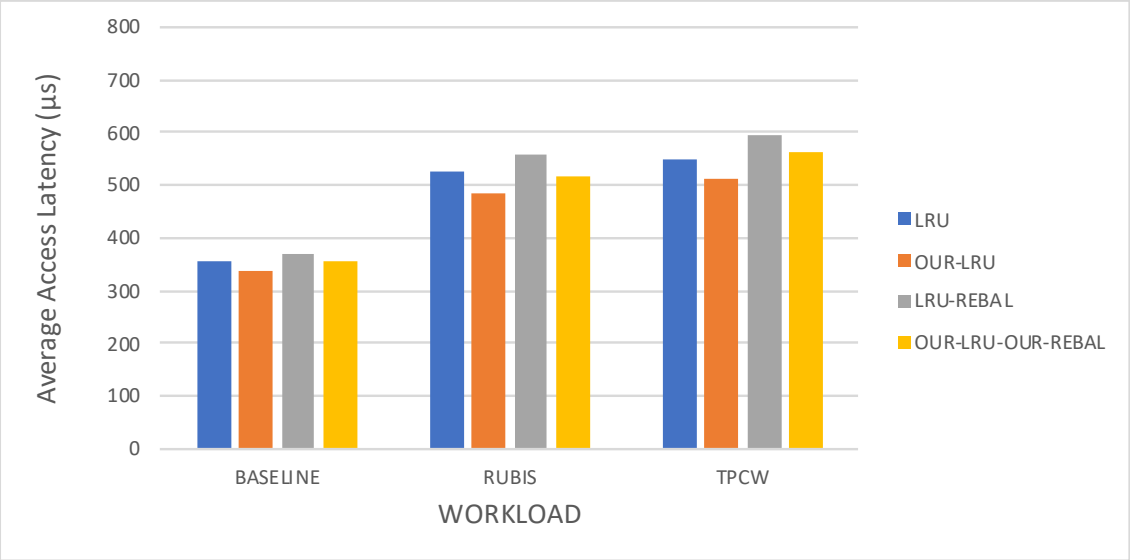

Figure 5.8: Average Access Latency (μs) for the multiple value workloads in Memcached 2GB RAM.

RUBIS workload.

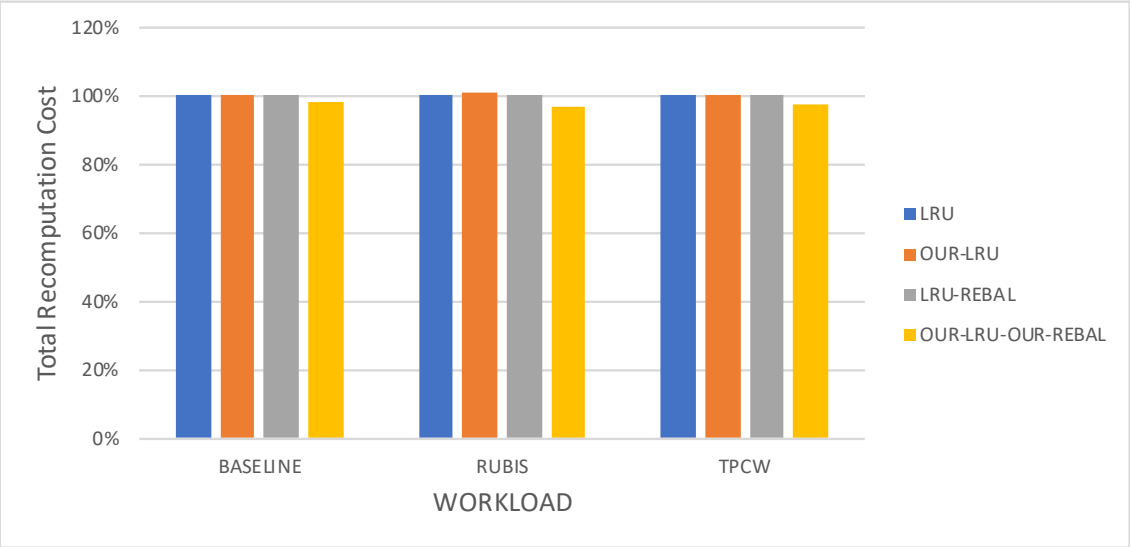

Figure 5.9: Average Access Latency (μs) for the multiple value workloads in Memcached 10GB RAM.

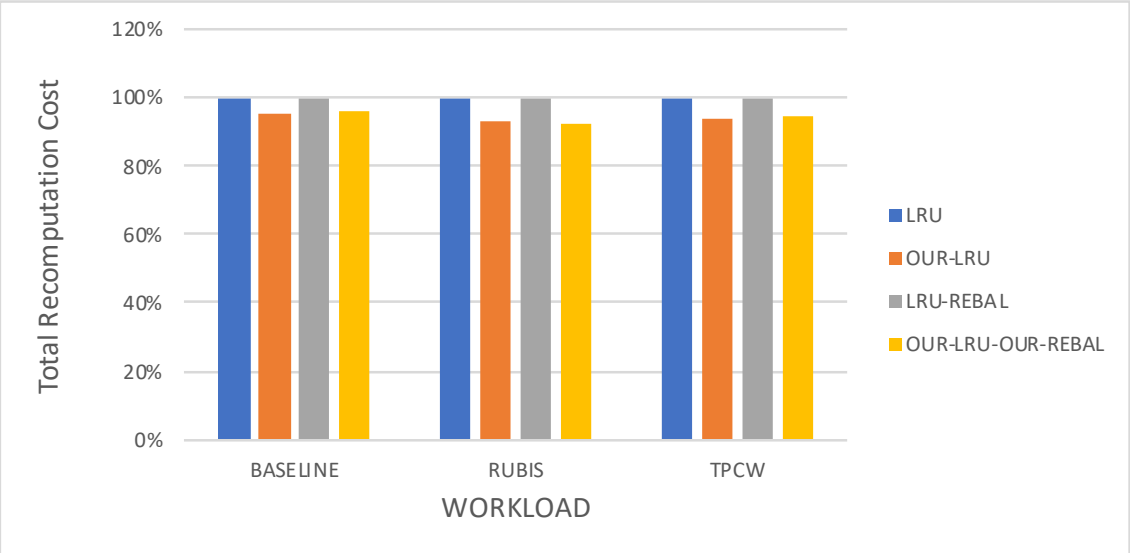

Figure 5.10: Average Access Latency (μs) for the multiple value workloads in Memcached 2GB RAM.

5.5.3 Discussion of the Results

In light of our objectives in this thesis, if the client can benefit more from a caching system that prioritizes more costly items? We can say that for a Memcached server of 10GB RAM it would not benefit as much as we expect since the average access time between our LRU and Memcached's LRU is practically the same. Although our replacement policy

did not have much impact in a cache size of 10GB, our rebalancing policy has shown an improvement of close to 3% in terms of average access latency.

On the other hand, we can see that for a Memcached server of 2GB our replacement policy can improve the average access latency and the total recomputation cost compared to the original replacement policy of Memcached, improving both up to 7%. Also, our rebalancing policy shows an improvement in the total recomputation cost in comparison between the original rebalancing policy of Memcached up to 8%.

Comparing our results with GD-Wheel, we noticed that GD-Wheels performance are better than ours. We can not directly compare their results with ours since they use a Memcached server of 25GB for their most experiments. However, GD-Wheel shows an improvement in the average read latency of about 33% and up to 53%, improving more than 46% than our proposed solution in the best scenario. GD-Wheel also reduces the total recomputation cost on average about 74% and up 90%, which comparing to our best scenario, GD-Wheel still improves more than 83%.

Taking into account what was said above some other approaches were made in the attempt of improving our results. We tried a new approach by having a new priority that was calculated by $h(p) = L_i + \frac{L_i * c(p)}{450} + c(p)$, being the priority of the key-value pair ($h(p)$) the same way as our initial proposal, but we added a weight that means that the highest cost can be worth an extra 10% of the inflation value(L_i). This way, we do not let the inflation value to overlap the cost value giving more priority eventually to more costly items. Since this attempt has brought similar results as the results presented above, we did not show them in this study. Since this approach did not bring better results, it could be interesting to increase the weight value of the priority to see if this would improve the results. For every slab class there is a reserved segment which contains 10% of the key-value pairs of the respective replacement policy, hence another strategy would be to tackle this value to see if some improvement is shown in the total recomputation cost, constraining more costly items in the reserved area would increase the total recomputation cost.

Chapter 6

Conclusion

6.1 Achievements

The work developed in this thesis has the objective of implementing and studying our approach of a cost aware memory management in Memcached. With our efforts, we also want to improve a key-value store by offering to the client applications an overall better performance. To do so, we start by understanding how different key-value stores work and how they can help us to scale applications, focusing on Memcached, a distributed in-memory cache. We analyzed, to know in more detail, how this distributed in-memory key-value store works and how its components interact among them, concluding that to take the values recomputation costs into account and for an improvement in its memory management there would be necessary a study on different memory management schemes, replacement policies and rebalancing policies. In this study, we discuss what replacement policies are and what factors can influence their decisions to evict items, describing how Memcached replacement policy works, and also giving special attention to cost-aware replacement policies like Greedy Dual-Size algorithm and GD-Wheel. At the end of our study, we go in more detail of the purpose of having object sizes rebalancing policies, describing the original rebalancing policy of Memcached and other approaches that can improve hit-rate or get more space for the most time consuming objects.

In this dissertation, it was introduced a replacement policy, easy to integrate into Memcached, which offers part of its memory for eventually storing more costly key-value pairs. Additionally, our proposal has parameters that can be adjusted to eventually improve our results. We also introduce a new rebalancing policy that takes the average cost per byte and frequency of the slab classes into consideration. It is also mentioned in this thesis how this replacement and rebalancing policy are implemented in Memcached.

To test our prototype, we built our experimental environment similar to GD-Wheels

with the purpose of future comparison of our work with other cost-aware approaches. This environment allowed us also to know how the new replacement and rebalancing policy of our prototype performed in terms of average GET, SET and Access latency, and to know how much total recomputation cost was reduced compared to the original Memcached. In order to analyze the efficiency of each policy, there are two types of workloads the single value and multiple value workload. The results have shown that our replacement and rebalancing policy do not show for a Memcached server of 10GB an improvement in the total recomputation cost and average access latency. On the other hand for a Memcached server of 2GB, our policy showed a slight improvement in the total recomputation cost and average access latency improvement up to 7% without rebalancing policy and up 8% with rebalancing policy in the best scenario. When comparing our results with GD-Wheel's, we noticed that GD-Wheel's shows a better improvement than our prototype on the total recomputation cost comparing to the old version of Memcached.

Concluding, the proposed objectives of this dissertation were accomplished. We were able to create our prototype, implementing our approaches in Memcached successfully. In terms of the client's access latency, our prototype showed some improvements for a key-value store of small memory, but it has not the improvements that we expected. Maybe if we work with some of the parameters, like the way the priority of the key-value pairs are calculated and the size of the reserved segment, it can possibly help our prototype to produce better results and scale better with more memory.

6.2 Future Work

Taking into account that our proposed solutions were developed in Memcached being in the stage of a prototype and the analysis of our results there are some changes that can be done to improve the total recomputation cost of key-value stores.

Our original proposed priority was based on Greedy Dual-Size priority. This priority relies on an inflation value to help newer key-value pair to keep up with key-value pairs with higher priority. This inflation value eventually will increase overlapping the cost value. In the attempt to overcome this, we tried to a new priority explained in the discussion of the results 5.5.3, but the results did not improve as we expected. It would be interesting to experiment if new weights that can give more priority to more costly key-value pairs.

We also plan to increment the percentage of our reserved segment to see if it could benefit the key-value store, by having more priority key-value pairs in the reserved area it could decrease the total recomputation cost. Another approach that we plan to do, that is related to the reserved segment, is to instead of having a reserved segment in each replacement structure in Memcached we would have a global reserved area with a global inflation value. This way the global inflation value is influenced by all the key-value pairs in Memcached, and the key-value pairs that will enter the global reserved segment would be the key-value pairs that are more costly in Memcached and not the most costly

key-value pairs in the slab class. Additionally, we plan on changing the way we set up the capacity of the reserved segments. The way we set up the capacity of the reserved segment is by passing as an argument in Memcached the size of how much key-value pairs that each reserved segment can store. We intend to change Memcached to be able to set up this argument dynamically by giving a percentage of the total memory of Memcached that we wish to allocate. Also, we pretend to study the relation between reserved space and global space, so that the best value can be used. Having into account the total memory, Memcached will allocate the passed that best percentage to the reserved segments.

Bibliography

[1] M. Abrams et al. *Caching Proxies: Limitations and Potentials*. Tech. rep. Blacksburg, VA, USA, 1995. URL: http://www.ncstrl.org:8900/ncstrl/servlet/search?formname=detail&id=oai%3Ancstrlh%3Avatech_cs%3Ancstrl.vatech_cs%2F%2FTR-95-12.

[2] C. Aggarwal, J. L. Wolf, and P. S. Yu. "Caching on the World Wide Web." In: *IEEE Transactions on Knowledge and Data Engineering* 11.1 (1999), pp. 94–107. ISSN: 1041-4347. DOI: 10.1109/69.755618.

[3] B. Atikoglu et al. "Workload Analysis of a Large-scale Key-value Store." In: *SIGMETRICS Perform. Eval. Rev.* 40.1 (June 2012), pp. 53–64. ISSN: 0163-5999. DOI: 10.1145/2318857.2254766. URL: http://doi.acm.org/10.1145/2318857.2254766.

[4] N. Beckmann and D. Sanchez. "Talus: A simple way to remove cliffs in cache performance." In: *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. IEEE. 2015, pp. 64–75.

[5] A. Blankstein, S. Sen, and M. J. Freedman. "Hyperbolic Caching: Flexible Caching for Web Applications." In: *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, 2017, pp. 499–511. ISBN: 978-1-931971-38-6. URL: https://www.usenix.org/conference/atc17/technical-sessions/presentation/blankstein.

[6] J. Bonwick. "The Slab Allocator: An Object-caching Kernel Memory Allocator." In: *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1*. USTC'94. Boston, Massachusetts: USENIX Association, 1994, pp. 6–6. URL: http://dl.acm.org/citation.cfm?id=1267257.1267263.

[7] P. Cao and S. Irani. "Cost-aware WWW Proxy Caching Algorithms." In: *Proceedings of the USENIX Symposium on Internet Technologies and Systems on USENIX Symposium on Internet Technologies and Systems*. USITS'97. Monterey, California: USENIX Association, 1997, pp. 18–18. URL: http://dl.acm.org/citation.cfm?id=1267279.1267297.

[8] J. L. Carlson. *Redis in action*. Manning Publications Co., 2013.

[9] D. Carra and P. Michiardi. "Memory partitioning in Memcached: An experimental performance analysis." In: *2014 IEEE International Conference on Communications (ICC)*. 2014, pp. 1154–1159. DOI: 10.1109/ICC.2014.6883477.

[10] D. Carra and P. Michiardi. "Cost-based Memory Partitioning and Management in Memcached." In: *Proceedings of the 3rd VLDB Workshop on In-Memory Data Mangement and Analytics*. IMDM '15. Kohala Coast, HI, USA: ACM, 2015, 6:1–6:8. ISBN: 978-1-4503-3713-7. DOI: 10.1145/2803140.2803146. URL: http://doi.acm.org/10.1145/2803140.2803146.

[11] J. Celko. *Joe Celko's Complete Guide to NoSQL: What Every SQL Professional Needs to Know About Non-Relational Databases*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013. ISBN: 0124071929, 9780124071926.

[12] A. Cidon et al. "Dynacache: Dynamic Cloud Caching." In: *HotStorage*. 2015.

[13] A. Cidon et al. "Cliffhanger: Scaling Performance Cliffs in Web Memory Caches." In: *NSDI*. 2016, pp. 379–392.

[14] G. DeCandia et al. "Dynamo: Amazon's Highly Available Key-value Store." In: *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*. SOSP '07. Stevenson, Washington, USA: ACM, 2007, pp. 205–220. ISBN: 978-1-59593-591-5. URL: http://doi.acm.org/10.1145/1294261.1294281.

[15] *Does Stack Exchange use caching and if so, how?* https://meta.stackexchange.com/questions/69164/does-stack-exchange-use-caching-and-if-so-how/69172#69172. Online; accessed 28 December 2017.

[16] B. Fitzpatrick. "Distributed Caching with Memcached." In: *Linux J.* 2004.124 (Aug. 2004), pp. 5–. ISSN: 1075-3583. URL: http://dl.acm.org/citation.cfm?id=1012889.1012894.

[17] *How We Made GitHub Fast*. https://github.com/blog/530-how-we-made-github-fast. Online; accessed 28 December 2017.

[18] X. Hu et al. "LAMA: Optimized Locality-aware Memory Allocation for Key-value Cache." In: *USENIX Annual Technical Conference*. 2015, pp. 57–69.

[19] *Internet Users*. http://www.internetlivestats.com/internet-users/. Online; accessed 03 February 2018.

[20] A. Khakpour and R. J. Peters. *Optimizing multi-hit caching for long tail content*. US Patent 8,370,460. 2013.

[21] R. Klophaus. "Riak Core: Building Distributed Applications Without Shared State." In: *ACM SIGPLAN Commercial Users of Functional Programming*. CUFP '10. Baltimore, Maryland: ACM, 2010, 14:1–14:1. ISBN: 978-1-4503-0516-7. DOI: 10.1145/1900160.1900176. URL: http://doi.acm.org/10.1145/1900160.1900176.

[22] C. Li and A. L. Cox. "GD-Wheel: A Cost-aware Replacement Policy for Key-value Stores." In: *Proceedings of the Tenth European Conference on Computer Systems*. EuroSys '15. Bordeaux, France: ACM, 2015, 5:1–5:15. ISBN: 978-1-4503-3238-5. DOI: 10.1145/2741948.2741956. URL: http://doi.acm.org/10.1145/2741948.2741956.

[23] *Libevent*. https://www.openhub.net/p/libevent___an_event_notification_library. Online; accessed 23 January 2019.

[24] S. Maffeis. "Cache Management Algorithms for Flexible Filesystems." In: *SIGMETRICS Perform. Eval. Rev.* 21.2 (Dec. 1993), pp. 16–25. ISSN: 0163-5999. URL: http://doi.acm.org/10.1145/174215.174219.

[25] *Memcached*. https://memcached.org/. Online; accessed 05 February 2018.

[26] R. Nishtala et al. "Scaling Memcache at Facebook." In: *nsdi*. Vol. 13. 2013, pp. 385–398.

[27] *Number of Facebook users worldwide 2008-2017*. https://www.statista.com/statistics/264810/number-of-monthly-active-facebook-users-worldwide/. Online; accessed 27 December 2017.

[28] *Redis*. https://redis.io/. Online; accessed 29 December 2017.

[29] *Redis Sharding at Craigslist*. https://blog.zawodny.com/2011/02/26/redis-sharding-at-craigslist/. Online; accessed 28 December 2017.

[30] A. Reveals. "Seconds as the New Threshold of Acceptability for eCommerce Web Page Response Times." In: *Press Reliase [Electronic resource].–September* 14 (2), p. 2009.

[31] M Seltzer. "Oracle nosql database." In: *Oracle White Paper* (2011).

[32] A. Silberschatz, P. Galvin, and G. Gagne. "Operating system concepts, 7th Edition." In: 2005.

[33] *Spymemcached*. https://github.com/couchbase/spymemcached. Online; accessed 21 January 2019.

[34] W. Stallings. *Network security essentials: applications and standards*. Pearson Education India, 2007.

[35] *Talk: Real-time Updates on the Cheap for Fun and Profit*. http://code.flickr.net/2011/10/11/talk-real-time-updates-on-the-cheap-for-fun-and-profit/. Online; accessed 28 December 2017.

[36] *Twemcache is the Twitter Memcached*. https://github.com/twitter/twemcache. Online; accessed 28 December 2017.

[37] *Twitter Company Statistics: Avarage number of tweets per day*. https://www.statisticbrain.com/twitter-statistics/. Online; accessed 27 December 2017.

[38] *Twitter: number of monthly active users 2010-2017*. https://www.statista.com/statistics/282087/number-of-monthly-active-twitter-users/. Online; accessed 26 December 2017.

[39] G. Varghese and T. Lauck. "Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility." In: *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*. SOSP '87. Austin, Texas, USA: ACM, 1987, pp. 25–38. ISBN: 0-89791-242-X. DOI: 10.1145/41457.37504. URL: http://doi.acm.org/10.1145/41457.37504.

[40] J. Wang. "A Survey of Web Caching Schemes for the Internet." In: *SIGCOMM Comput. Commun. Rev.* 29.5 (Oct. 1999), pp. 36–46. ISSN: 0146-4833. DOI: 10.1145/505696.505701. URL: http://doi.acm.org/10.1145/505696.505701.

[41] *Web Protocols and Practice: HTTP/1.1, Networking Protocols, Caching, and Traffic Measurement*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001. ISBN: 0-201-71088-9.

[42] *What is an In-Memory Key-Value Store?* https://aws.amazon.com/nosql/key-value/. Online; accessed 24 January 2018.

[43] N. Young. "Thek-server dual and loose competitiveness for paging." In: *Algorithmica* 11.6 (1994), pp. 525–541. ISSN: 1432-0541. DOI: 10.1007/BF01189992. URL: https://doi.org/10.1007/BF01189992.

[44] N. Zaidenberg, L. Gavish, and Y. Meir. "New caching algorithms performance evaluation." In: *2015 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*. 2015, pp. 1–7. DOI: 10.1109/SPECTS.2015.7285291.

[45] V. Zakhary, D. Agrawal, and A. E. Abbadi. "Caching at the Web Scale." In: *Proc. VLDB Endow.* 10.12 (Aug. 2017), pp. 2002–2005. ISSN: 2150-8097. DOI: 10.14778/3137765.3137831. URL: https://doi.org/10.14778/3137765.3137831.

[46] H. Zhang et al. "In-Memory Big Data Management and Processing: A Survey." In: *IEEE Transactions on Knowledge and Data Engineering* 27 (2015), pp. 1920–1948.

[47] *Zynga memcached module for PHP*. https://github.com/zbase/php-pecl-memcache-zynga. Online; accessed 28 December 2017.