Vasco Samuel Rodrigues Coelho

Bachelor of Computer Science and Engineering

Study and optimizion of the memory management in Memcached

Dissertation plan 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

February, 2018

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, an user wont 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 bringing the data closer to users decreasing the number of access to the slower data storage's. The first implementation of Memcached, 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 an hit rate around 92%, reducing the number of accesses to the database and response time considerably.

In this research a study will be performed in order to evaluate where Memcached can be improved, performance-wise, by addressing its memory management scheme. To do so we first take a deeper look into key-value stores and how they work as a web cache. We later analyze how Memcached works, respectively how its memory management, replacement and rebalancing policy occur. Since this components are in direct contact with the memory management, in this study we analyze and explore how each component work and how can different approachs benefit Memcached's performance.

Keywords: Web cache; Memory management; Memcached

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 instancias do Memcached em dez *hosts* únicos, fazendo *caching* dos 30GB de dados dos objectos mais populares atingindo uma taxa de sucesso de acerca de 92%, reduzindo consideravelmente o numero de acessos para a base de dados e o tempo de resposta.

Esta investigação tem como objectivo fazer um estudo para encontrar maneiras de aumentar a performance do Memcached, endereçando a seu esquema de gestão de memória. Para que tal aconteça, primeiro olhamos com mais atenção para os repositórios chave-valor e como é que estes funcionam sendo uma cache web. Em seguida será feita uma analise ao funcionamento do Memcached, respectivamente à sua gestão de memória, políticas de substituição e rebalanceamento. Uma vez que estes componentes estão em contacto directo com a gestão de memória, neste estudo vamos analisar e explorar como cada um dos componentes funciona e como é que diferentes abordagens podem beneficiar a performance do Memcached.

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

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 grown rapidly, possibly because the Web is relatively inexpensive to use and accessing data through it is faster than any other means [39].

Over time there has been a substantial increase of users in social media networks [27, 37]. In the case of Twitter's users, an average of 58 million tweets is sent every day [36]. This, results in millions of page requests that the company needs to deal with high speed to satisfy user needs. Studies have shown that an user wont wait no more than three seconds for a web page to load before abandoning the site [30]. The traditional disk-based storage systems aren't enough to handle this situation due to the fast data access required by users, degrading the overall performance significantly [47]. 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. [46].

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 important 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, 35, 48]. Likewise, Github, Flickr, Stack Overflow and others use Redis as their distributed caching system [15, 17, 29, 34]. 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 an 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, 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 a good predictability on requests. However, like Zaidenberg et al. [45] 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 more 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 a lot of 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's 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 a 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 can't be easelly applied to Memcached for efficiency reasons 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.

Memcached memory management is designed based on the slab allocator of Linux [6]. The memory is partitioned into classes, called slab classes. The slab classes are composed of a group of slabs. Each Slab it's divided into equal size blocks of memory called chunks, these chunks store objects of a specific ranged size, the range differs from class to class.

Memcached uses a rebalancing policy based on the eviction rates between slab classes. Rebalancing polices 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 rebalacing policy may have influence on having better results. Some researchers [10, 12, 13] have stated that by using dynamic memory algorithms, Memcached can achive better results. We believe that by mixing these dynamic memory allocation 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, incresing 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 read latency significantly.

1.2 Problem Statement

Cost-Aware Replacement policies are arising for the purpose of tackling systems that benefit from caching non-uniform cost based objects in their key-value stores. These policies can't be directly implemented 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. 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 pages taking into consideration the average cost of each class, making possible to combine it with the GD-Wheel algorithm. However, the authors don't 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 pages from the lowest priority slab class might not always be beneficial. There could be some classes where the reallocation of their pages doesn't increase the miss rate as much as if we take constantly from the low priority queue.

1.3 Objectives

The aim of this dissertation is to study how cost aware memory management schemes can influence cost based replacement policies in achieving better results on Memcached. This way we can see the importance of memory management when new factors of the object are taking into account e.g cost and size.

Our objective is to study the use of a dynamic cost-aware memory allocation with cost-aware replacement algorithms in Memcached. The study will allow us to compare if the Cost-Aware Rebalancing Policy created by Conglong et al. had a big influence on their results. We can also study if a new cost-aware memory management scheme will achieve better results, showing that their Cost-Aware rebalancing policy isn't optimal just by continuously moving pages from the lower priority slab class.

1.4 Expected 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 replace its default replacement and rebalancing policy. The default replacement policy of Memcached would be replaced with a cost-aware replacement policy and the default rebalancing policy would be replaced for a new dynamic cost-aware rebalancing policy. This new dynamic cost-aware rebalancing policy would be the combination of two algorithms that were never combined before. Additionally, a prototype will be made to simulate the model of the new system that was planned. Consequently, a critical analysis will be made from the gathered data provided by the prototype.

1.5 Document Organization

The next chapters of this documenten 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 focus on approaches that can be done to optimize Memcache's memory management, replacement and rebalancing policies.
- **Chapter 3** shows our work plan along the elaboration period of our thesis.

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 haves one pair for the 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**: searchs 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, they are 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 that the SQL databases can't provide. Unlike SQL databases, key-value stores don't 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 don't rely on a formally defined data types, allowing any kind of data in the system.

SQL databases use normalization of data. To avoid replication, redundancy and anomalies, 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 SQL databases can efficiently manage its data, but in exchange performance is lost when processing the data because sometimes the normalization process can become complex and it will involve a lot of this tables to create a logical meaning. Key-value stores don't usually work with data relations, therefore normalization is rarely used in this NoSQL data model. Taking all this in consideration, key-value stores have a simple structure and their query speed is higher than relation databases. The possibility of not having a schema or normalization enable key-value stores to easily support distributed mass storage and high concurrency. Searching and modifying data operations are efficient and can deal with high data values and disks, making key-value stores good for high read-applications by scaling with more resources. Also the system can't scale to deal with huge volumes of data or requests by joining more hosts and using data partitioning and replication.

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

Web cache is the 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. Caches contain a subset of data that is also stored in the database. The goal is to keep the most used web documents in the cache to avoid taking more time to the database. 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 an highly concurrent fashion provide the opportunity for applications like Amazon ElasticCache [41] to be optimized 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. When a match occurs, the key-value store will return the value associated to the requested key. Likewise, when the cache doesn't find what was requested, it will return a null value to the application (step 4).

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

Depending on the result of the key-value store, if a value was returned, the application 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).

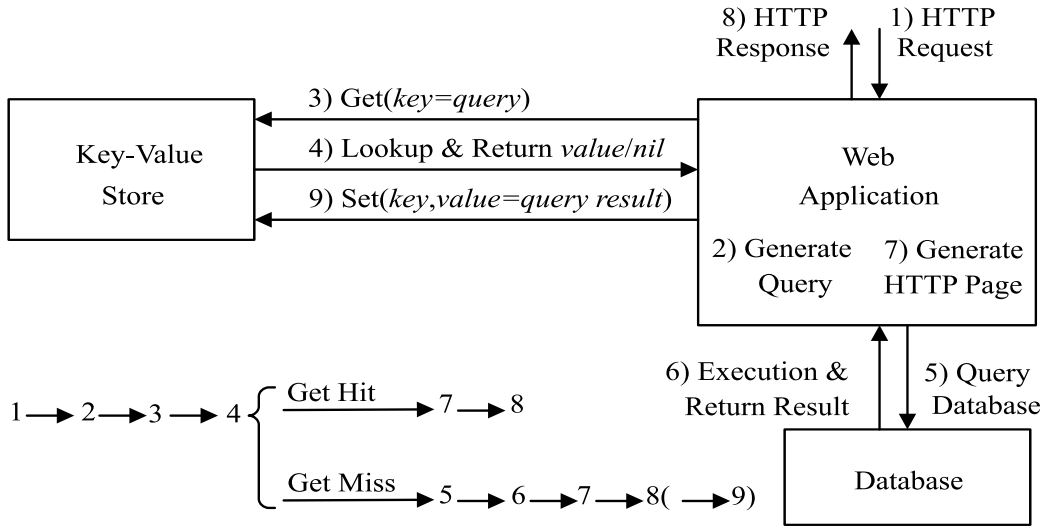


Figure 2.1: Using a Key-Value Store as a Database Query Cache.(taken from [22]).

2.2 Memcached

Memcached [16, 25] is a distributed in-memory cache supporting arbitrary strings as its data type. It is multi-threaded, making it ideal for read-only data. Memcached presents to the user an optimised dictonary 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, 35, 48].

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 [33] 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 doesn't provide redundancy. 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 hipotetical big cache that is consistantly dealing with the same set of keys. Each Memcached Server relies on its Memory allocation, Replacement, and Rebalancing policies, to achieve his greatest local performance. Due to the way Memcached allocates his memory it can achieve constant time complexity on his add, get, set and flush operations. This is an important aspect because the functions should take as much time to execute when the cache as just one item or when it is full.

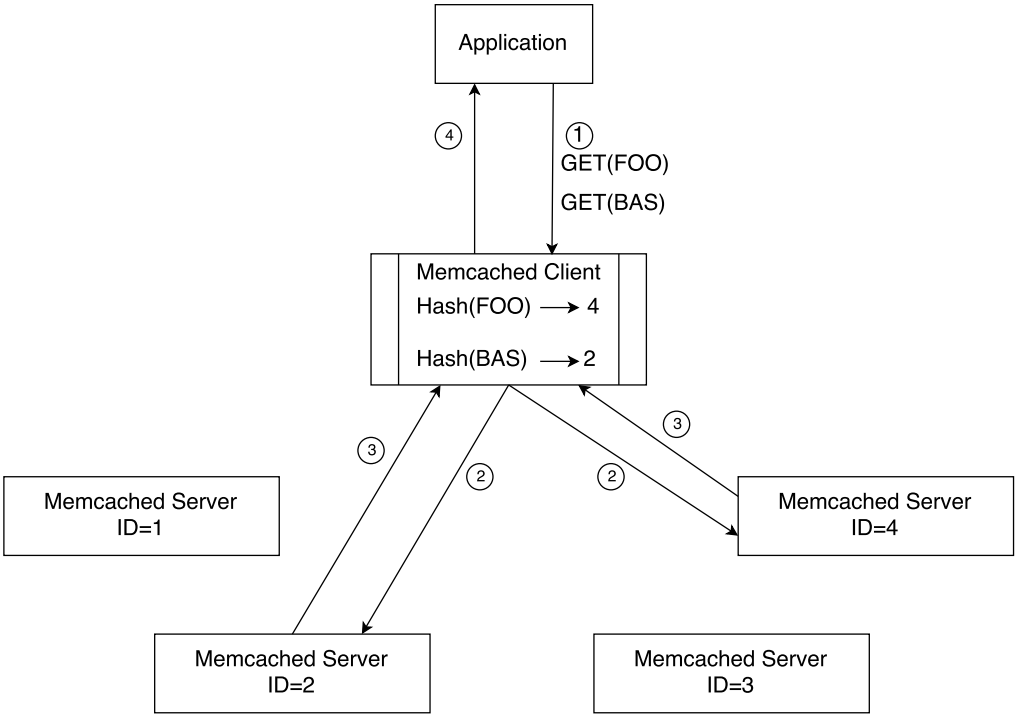


Figure 2.2: How Memcached works.

2.2.1 Memory allocation

Memcached, for its memory management uses a technique called the Slab Allocator [6]. The Slab Allocator is a simple, fast, and space-efficient memory allocator also used in Linux kernel. The allocation and freeing of objects are common actions used in the kernel, so it's essential that the memory allocator is fast for a better performance of the kernel. Constructing and destroying objects takes a considerably amount of time, exceeding the time of allocating and freeing memory of objects [6]. To address this problem object caching is used as a technique to deal with objects that are frequently used. With slab allocations, the **slab** its the primary unit of currency, representing blocks of memory that usually are $1MB$ each. A slab is sliced into **chunks** of the same size. The slabs are store in partitions of the memory depending on their chunk size, this partitions are called **slab classes**. Slabs that are in different Slab classes contain chunks of various sizes.

Memcached initially starts to pre-allocating its memory by dividing the memory into slab classes. The total available memory is allocated to classes in a slab-by-slab way [10]. By pre-allocating its memory into the form of slab classes, Memcached avoids internal fragmention [42], because it only allocates memory that its need. Internal fragmentation happens when you allocate a block larger than needed. When you allocate an object in that block, there will be a space that can't be used because the remaining space doesn't allow to store another object.

An object is allocated in the class that has chunks with sufficient space to contain it. When an object wants to be stored, and there are no available chunks, Memcached will first check if there is available memory in the Slab class. If there is available memory in the class a new slab is created, else an existing object have to be evicted in order to give room to the new object. Memcached by allocating the objects in the respective classes of their size, it ensures a controlled external fragmentation [42]. External fragmentation happens when free blocks of memory are available for allocation, but they can't be used, since they are fragmented into small blocks and because of that they can't hold objects of the sizes requested. Since each class as a size range to store objects, the max external fragmentation can up to the difference between the maximum and minimum of the range.

2.2.2 Replacement policy

When it comes to eviction decisions of objects, Memcached uses a trivial LRU algorithm to pick the object that must leave [45]. In essence, Least Recently Used (LRU) is an algorithm which evicts the object least recently used in the cache. It can also be called recency.

In this context when we say that the object is least used in the cache, we mean that an object is the least requested among the other objects.

2.2.3 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 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. If we take into account 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.

2.3 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 management dedicated to the kernel. In the first case for a program to execute its necessary 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 optimize the performance of overall system. As for the kernel, the allocation and freeing of objects are common operations, therefore memory management needs to handle the memory in an efficient and quick way, 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, showing more quickly to the final user what he wishes. In-memory key-value stores are used to store the data more used, avoiding the intensive access to the 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's 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 will be needed to access the database or remote resources.

In the following subsections, we are going to explore some memory management schemes that exist.

2.3.1 Fixed-size blocks allocation

A. Silberschatz et al [32] said that the easiest method for allocation memory is to divide the memory into fixed-sized partitions. Fixed-size memory allocation or fixed-partition scheme is essentially 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 the block is allocated, leaving the rest of the unused memory for future requests. Only one process can occupy a block so if we have a block of $5KB$ and two processes need to allocate $4KB$ and $1KB$ respectively, the process of $4KB$ will occupy the $5KB$ because it was first leaving $1KB$ of unused memory. Even though there's $1KB$ of available memory in the block, the second process can't allocate that memory, creating a problem called internal fragmentation. When a process terminates, it releases its memory, giving the opportunity to other processes to allocate that block.

Eventually, the large enough blocks are going to be occupied and the processes that are in need of memory must wait in a queue for there turn when a large enough block is released. 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's 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, but there are some memory management schemes that 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 in 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 size of the pages is defined by the hardware 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 page isn't completely filled by a process its still allocated to that same process, making it unusable to other processes and creating internal fragmentation. **Page table**, as we can see from figure 2.3 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's possible to determine where is the page going to be stored in the physical memory.

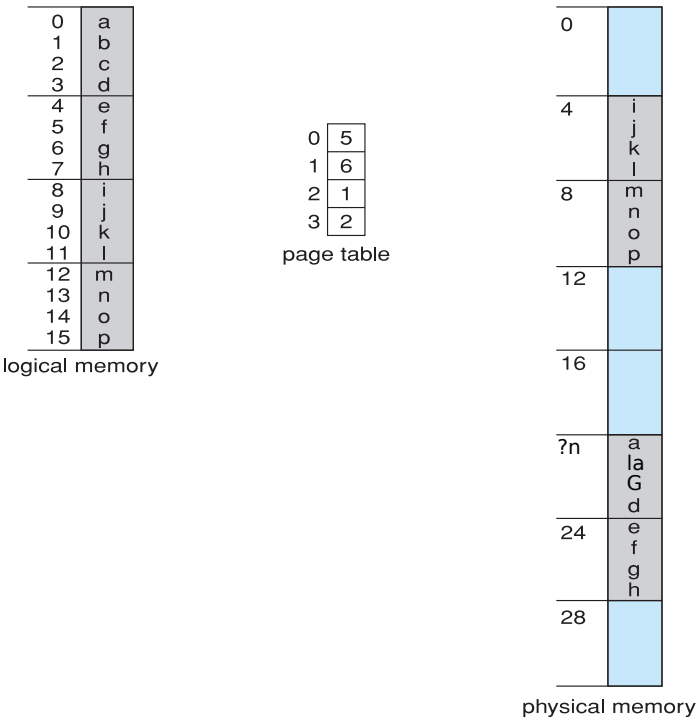


Figure 2.3: Paging model of logical and physical memory. (taken from [32]).

In the next example based on A. Silberschatz et al example 2.3 we are going to describe how pages are assigned to physical memory. Lets 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 the 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 $((2*4)+0) = 8$.

2.3.2 Slab allocation

The slab allocator [6] is an object-caching memory allocator used in 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,that object is destroyed and memory is freed. Its expensive to create and destroy objects because this operations require the allocation and freeing of memory. Therefore slab allocation addresses the problem by caching this objects and its free or in-use state, avoiding this way the of frequent allocation and freeing of objects memory.

The Slab allocator works with units called slabs. A slab represents one 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 2.4 we can

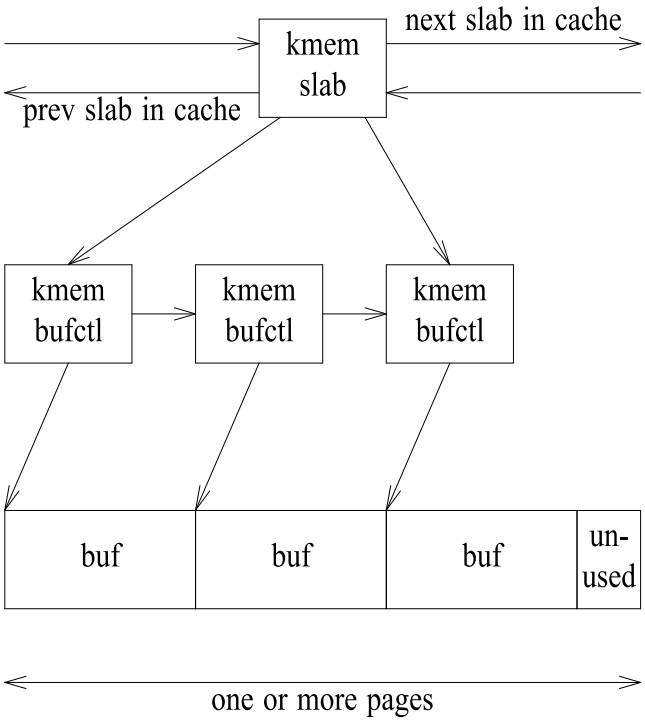


Figure 2.4: Slab logical layout. (Taken from [32]).

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 freelist 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 freelist of available buffers. This originates a two-level freelist structure, that helps the memory reclaiming because it just simply needs to unlink a slab instead of unlinking every buffer from the cache's freelist.

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

- **Reclaiming unused memory is trivial.** As we mention above, the cache presents a two-level freelist 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 freelist and update a reference count. Since the freelist 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 freelist) 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 10 buffers, leaving an unused 96 bytes. This results in 9.6 bytes wasted per slab, that represents a 2.4% internal fragmentation.

2.3.3 Segmentation Memory Management

Segmentation is 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 o 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 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 its purpose in the program. Each segment has a name or number and a length.

Despite 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 in 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 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 operating system which means the logical addressing attempt is beyond end of segment.

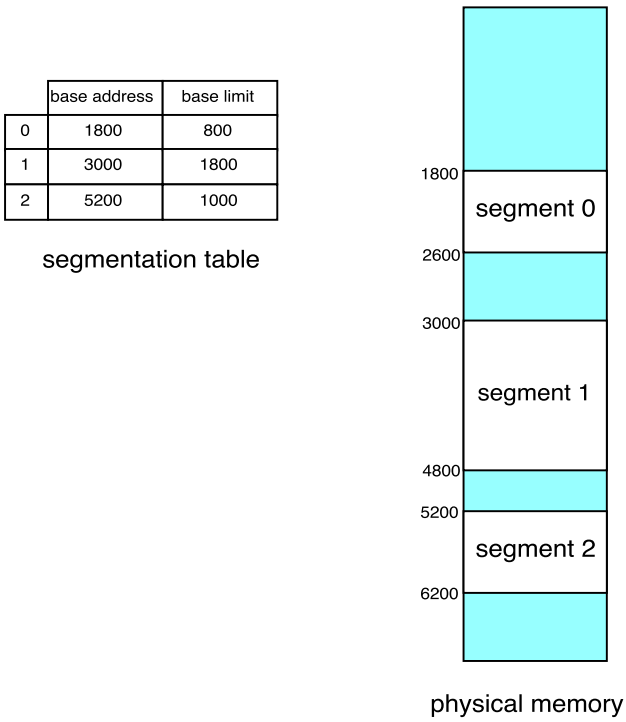


Figure 2.5: Example of segmentation. (inspired by [32]).

2.3.4 Summary

Memory management schemes were built to handle memory in a way that process 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 isn't visible to the user and is controlled by the operating system.

Segmentation on the other hand of paging, its a memory management scheme that is visible to the user and segments don't follow a specific fixed size. Segmentation works similarly like paging, it also as a segmentation table, which indicates where the segmentation starts and ends. Any process can allocate a segment if there's available memory because a segment doesn't follow any specific size. For this reason, segmentation avoids internal fragmentation. But on the other hand, segmentation has more external fragmentation due to the fact that freed segments leave empty spaces that fragments 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 its common to insert and deleting 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.

2.4 Replacement Policies

When the cache becomes full of objects and it wants to store new objects, a replacement policie is performed. The replacement process is performed by evicting older objects in order to make space for the new 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 [40] 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 time the object was requested or accessed.
- **size**: size of the object.
- **cost**: this cost is referred to the latency of fetching an object from origin source server(normally database) or to recalculate 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 aging process to removed objects that were popular once but aren't anymore).

There are different replacement policies derived from this 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 Rencently Used (LRU) policy [23]. The LRU policy assumes that the least recently accessed as the least probability to be accessed in the future. The advantage of using this algorithm is that incorporates a aging mechanism in objects e.g if a 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 recent stored objects to scale their priority in cache. The disadvantage of using LRU is that sometimes there are popular objects that can collide with recent objects that can have more priority just 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 [23], 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. Ignoring the recency can bring this policy some disadvantages as new objects don't have time to build up their frequency to match older objects. Also, an object that was popular once and received a lot of 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 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, more smaller objects can fit in the space left by the larger object. However, some websites can benefit from larger objects because the objects are harder to compute and therefore its 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 and cost: Greedy Dual [7], LRU-Min [1], PSS [2].

Cost aware policies. Web objects are constantly being recomputed. Some, in order to be recomputed, need to gather information that is stored in databases. The latency 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 important in some systems. Since most caches use rencency as their decision 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, their size and recency to decide which objects should be evicted in the cache.

Modification time aware policies. When modification time is taken into consideration, what is really done is to consider the period time since the object was last modified. Although, we can consider the last time since the object was accessed as the eviction decision, it can bring up several disadvantages. The major disadvantage is that objects that are frequently popular can be evicted because the time since the object was last accessed is bigger than the time of the other less requested objects. Instead this factor is used as a aging mechanism and its combined with other factors to tackle there 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 bases on that rank to make their eviction decisions.

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

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

In the follow subsections we are going to explore some replacement policies that could benefit from cost factors.

2.4.1 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 [44]. Their solution uses a priority queue to store every priority of each cached object. Every cached object has a priority value H, and the queue has a global inflation value L. 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's no more room to store it, the global variable L is updated, and it's 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($O(\log n)$.).

2.4.2 GD-Wheel Replacement Policy

The GD-Wheel policy is a combination of two algorithms: the Greedy Dual Algorithm [7] and Hierarchical cost wheels that is inspired by Varghese and Lauck's Hierarchical Timing Wheels [38]. 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 its possible to restrict the priority range, constant time complexity is achievable. They reduce the time complexity by leveraging the Greedy Dual algorithm with Hierarchical Cost Wheels.

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 doesn't 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's 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, objects are inserted or moved to the respective queue by adding the objects cost 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 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.

2.4.3 Summary

There are a lot of 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 a attempt to optimize more complex systems. Although many systems use LRU as their replacement policy, we saw that there are some more complex systems that

could benefit if they also took into consideration the cost of latency as their main factor of decision. There are few replacement policies that use cost as their decision 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 deeper 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 latency is an important factor in the system.

2.5 Rebalancing Policies

Memcached allocates its memory in a slab-by-slab ways. This means that Memcached allocates its memory in the respective slab classes according to the received requests. A slab class that is more requested will have more memory allocated than other less requested slab classes. When the available memory of Memcached is assigned to the slab classes, it will always remain associated to the respective classes, which means that when the request distributions change in Memcached, slab classes may not have enough memory and their queues may be too short. This happens because the slabs where allocated taking into account the old request distribution. If the requests stay uniform the hit ratio won't change much and consequently the performance will be the same. However there are systems whose request aren't uniform and change over time. These non-uniform requests degrade performance because there are classes with low memory allocated due to the fact that they were low requested in the past. Now if these classes became popular, Memcached is forced to do more frequently evictions, because those classes become full quickly. This problem is called slab calcification [9, 18]. To tackle the slab calcification problem some approachs 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 a lot of requests.

Rebalancing Policies. These type of policies will auto move slabs between slab classes. The movement of slabs to other slab classes is restricted by the policies conditions. As an example, Twemcache[3] [35] has a set of rebalancing policies to avoid the slab calcification.

- Random slab eviction: For every SET operation, if there isn't 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.

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

2.5.1 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 in 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 a number of 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 won't have time to build up to migrate to other classes.

2.5.2 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 can't be applied directly into Memcached for efficiency reasons, because Memcached has a particular memory management scheme that works specifically with LRU policies. 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

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

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 weren't 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's 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 done to avoid the saturation of 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].

2.5.3 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, 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 has knowledge of the entire hit rate curve.

Cliffhanger with the knowledge of the local hit curve, it 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 described 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 2.6 you can see an example of how Talus works.

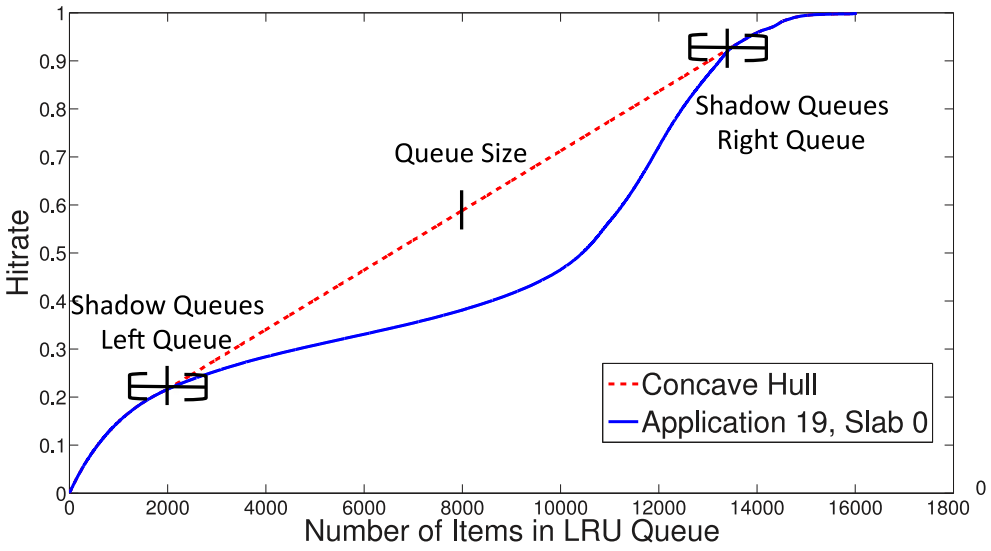


Figure 2.6: Talus example.(taken from [4]).

Talus needs all the hit rate curve to determine if there is a performance cliff, and since Cliffhanger is based on local knowledge there's 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 improved significantly the hit rate of a data center memory cache.

2.5.4 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 rebalancing 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 can't dynamically adapt efficiently to different application access patterns. A. Cidon et al. [12] said in their paper, that one-size-fits-all cache behavior 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 2.7. 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 to solve 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 of 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.

$$\begin{aligned} \underset{m}{\text{maximize}} \quad & \sum_{i=1}^{s} f_i h_i(m_i, e) \\ \text{subject to} \quad & \sum_{i=1}^{s} m_i \leq M \end{aligned}$$

Figure 2.7: Optimal memory allocation equation of Dynacache. (Taken from [12]).

2.5.5 Summary

Memcached allocates its memory in a slab-by-blab 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 aren't being used so 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 wasn't 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 weren't 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 doesn't 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 2.7. 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.

2.6 Critical Analysis

We believe that Memcached has the potential to improve its overall performance helping systems to achieve better hit rates in their caches. In order to do it, there are three ways we can explore in Memcached: the memory management, the replacement policy and the rebalancy policy.

Memory management Currently Memcached uses slab allocations to cache their objects. With this approach severe external fragmentation is unlikely to happen due to the continuous pre-allocation of memory, however it still occurs some internal fragmentation even if its minimal in the blocks. A possible study can be done to observe how different memory management schemes can affect the performance on Memcached. In more detail, we can explore if different memory management schemes can reduce the internal fragmentation in Memcached, making more memory available to store more objects, and we can analyze if this space can contribute to the overall performance of Memcached.

Replacement and Rebalancing policies In our research we noticed that the characteristics of the objects like the cost of recomputation are important factors that need to be taken into consideration when managing the cache. Replacement policies like GD-Wheel use as their decision factor the cost of recomputing an object when evicting an object, and can achieve amortized constant time complexity per operation if they restrain the number of hierarchical cost wheels. Conglong Li and Alan L. Cox [22] integrated GD-Wheel with their cost-aware rebalancing policy for efficiency reasons, since Memcached, default rebalancing policy works taking into consideration the LRU policy for evicting objects. We notice that their cost-aware rebalancing policy presents one problem, the policy 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 won't have time to build up to migrate to other classes. Taking all this into consideration, we can explore this problem and address it by combining some rebalancing algorithms like Carra's et al. [10] Cost-based Memory Partitioning and A. Cidon et al. [13] Cliffhanger. If we

take advantage of Cliffhanger's performance cliff solver and Carra's Cost-based Memory Partitioning algorithm we might achieve a better memory management than Conglong Li and Alan L. Cox cost-aware rebalancing policy. Afterwards, we can study the affects of this combination with GD-Wheel and analyze if the results obtained in GD-Wheel were due to the replacement policy or due to the new memory management scheme.

Chapter 3

Work Plan

In this chapter we present the work plan for this thesis. Table 3.1 describes in detail each planed task and the following Ghant chart illustrates the timeline for each of them.

#	Task	Description
1	**Discuss next phase steps**	A study of possible approaches to address the problem will be made, taking into consideration the performed related work. Following the study, a solution will be planned.
2	**Design of system model**	The design and as stated in the objectives will be made according to what was defined in the planning for the solution.
3	**Setup and configuration of Development environment**	Installation followed by its setup and configuration of all required software to implement the system model.
4	**Development of the system model**	Implementation of the solution through a prototype, according to what was defined in the system model.
5	**Experimental Evaluation**	Experimental verification of the prototype in order to understand the approach made.
5.1	Perform testbed of the system model	Setup of the environment for testing the prototype.
5.2	Integration tests and adjustments of the prototype	Checking if the behavior of the prototype is as expected.
5.3	Perform benchmarks of the results	Measurement and comparison of the results.
5.4	Critical analyze of the experimental results	Rigorous analysis of the results obtained through the tests performed in the prototype. Study and interpretation of benchmarks.
6	**Completing dissertation report**	Adjustments of the state of the art and complementing the dissertation report with the system model and the experimental evaluation.
6.1	Rewriting the state of art	Make adjustments to the related work, focusing more on the approach that I'm going to follow.
6.2	Partial writing of dissertation	Report the design of the system and the partial results.
6.3	Revision of final dissertation report	Evaluation of the final report and correction of errors that may appear.

Table 3.1: Description of work plan tasks.

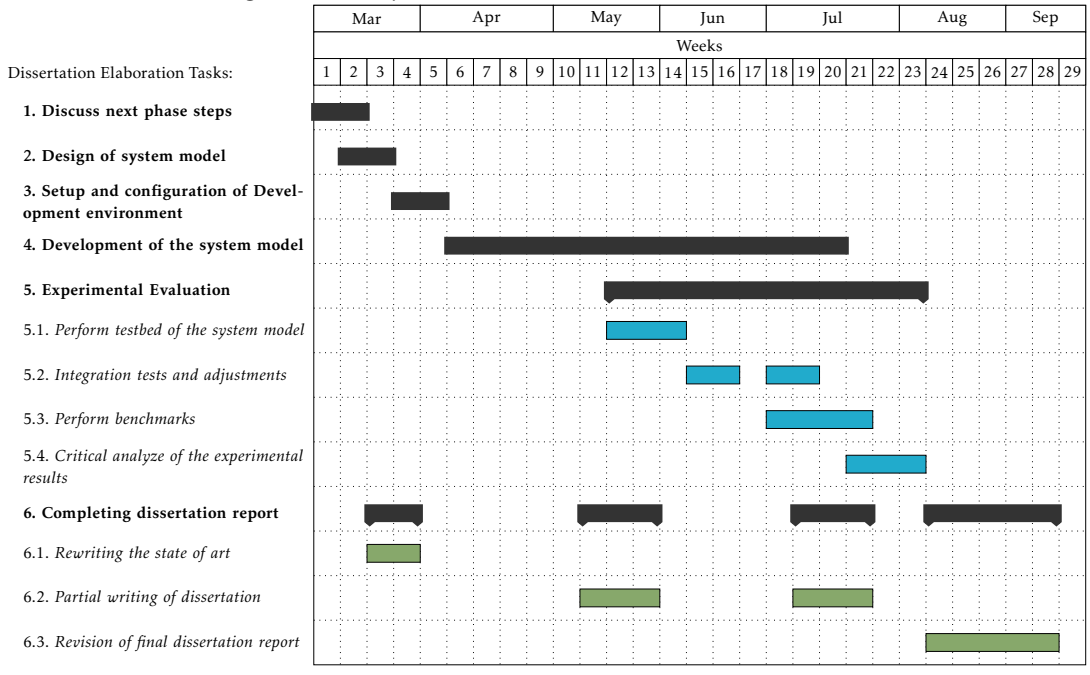
Although we describe the tasks in the table the following task is described in more detailed:

Experimental Evaluation. To do the evaluation we can use ycsb-memcached[43] and memaslap[24]. With this frameworks, its possible to generate configurable workloads for Memcached. Our evaluation is based on the GD-Wheel [22] evaluation, because we intend to use the same type of objects.

For all workloads we will use keys and values of the same size and the GET-to-SET ratio will be as used in facebook 30:1, because a considerably amount of SET's its needed in order to study the memory management of Memcached. As an example the follow work loads are based on GD-Wheel's:

- Workload 1: Has three groups of cost variation with different cost distribution.
- Workload 2 and 3: Uses the same cost groups and the cost variation from RUBiS and TPC-W, respectively.
- Workload 2: Has different size objects.
- Workload 3: Uses the same cost for all objects.
- Workload 4: Uses a random cost distribution.

For the different workloads we want have results for the Average Application Read Access Latencies, we want to see the total time it takes to access the database; Hit/Miss Rate; The cost for GET/SET operations; The percentage of memory used and; the number of times the rebalancing of memory occurs.



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] 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.

[24] *memaslap - Load testing and benchmarking a server*. http://docs.libmemcached.org/bin/memaslap.html#details. Online; accessed 15 February 2018.

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

[34] *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.

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

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

[37] *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.

[38] 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.

[39] 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.

[40] *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.

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

[42] P. R. Wilson et al. "Dynamic storage allocation: A survey and critical review." In: *Memory Management*. Ed. by H. G. Baler. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 1–116. ISBN: 978-3-540-45511-0.

[43] *Yahoo! Cloud Serving Benchmark*. https://github.com/brianfrankcooper/YCSB/wiki. Online; accessed 15 February 2018.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

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