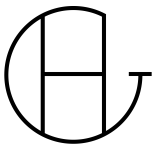

HoGent

Faculteit Bedrijf en Organisatie

Research of caching strategies in mobile native applications using external data services

Frederik De Smedt

Scriptie voorgedragen tot het bekomen van de graad van
Bachelor in de toegepaste informatica

Promotor:
Joeri Van Herreweghe
Co-promotor:
Jens Buysse

Instelling: —

Academiejaar: 2015-2016

Tweede examenperiode

Faculteit Bedrijf en Organisatie

Research of caching strategies in mobile native applications using external data services

Frederik De Smedt

Scriptie voorgedragen tot het bekomen van de graad van
Bachelor in de toegepaste informatica

Promotor:
Joeri Van Herreweghe
Co-promotor:
Jens Buysse

Instelling: —

Academiejaar: 2015-2016

Tweede examenperiode

Abstract

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Voorwoord

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetuer.

Suspendisse vel felis. Ut lorem lorem, interdum eu, tincidunt sit amet, laoreet vitae, arcu. Aenean faucibus pede eu ante. Praesent enim elit, rutrum at, molestie non, nonummy vel, nisl. Ut lectus eros, malesuada sit amet, fermentum eu, sodales cursus, magna. Donec eu purus. Quisque vehicula, urna sed ultricies auctor, pede lorem egestas dui, et convallis elit erat sed nulla. Donec luctus. Curabitur et nunc. Aliquam dolor odio, commodo pretium, ultricies non, pharetra in, velit. Integer arcu est, nonummy in, fermentum faucibus, egestas vel, odio.

Contents

1 Introduction **4**
1.1 Problem definition and research questions 5

2 Methodology **6**

3 Basics of caching **8**
3.1 Basic cache performance metrics 8
3.2 Paging 9
3.3 Locality principle 10
3.3.1 What is the locality principle 10
3.3.2 Interpreting spatial-temporal distances 10
3.4 CRUD 12

4 Cache replacement algorithms **13**
4.1 RR 13
4.2 FIFO 14
4.3 LRU 15
4.4 MRU 16
4.5 LFU 18
4.6 LIRS 19
4.7 Bélády's algorithm 21

5 Conclusion **23**

Chapter 1

Introduction

The definition of mobile has changed a lot in the last few years and the expectations of mobile applications continue to rise. Mobile applications should work on a myriad of devices and screen resolutions, and should take battery efficiency into account. Furthermore is an application no longer really considered mobile if it isn't available without a stable internet connection. This makes research about caching in mobile applications really interesting since it can solve multiple problems. Caching of the results of expensive operations, e.g. an internet connection, will allow the system to avoid many of these operations, reducing battery consumption, and can these results be used when a (temporary) connection loss would occur.

Lots of IT companies already implement such a system:

- The Facebook-app still shows several posts and pictures when you have no internet connection;
- Third-party API's, such as Picasso (Android) and SDWebImage (iOS) retrieve images from the web and have an embedded caching system;
- Twitter allows you to change account settings offline, which are later synchronized with the backend when a connection is established[1].

[1] In this example we are talking more about persistent data, however it is used to store temporary data that can later be forgotten once it is synchronized with the backend. Just like a cache containing business data.

1.1 Problem definition and research questions

However mobile caching is becoming more and more common, there is no real framework on which they all rely. Instead they each have to think about how to implement caching and have to invest in discovering caching strategies and inventing a good caching implementation.

They can decide to ignore the caching problem and focus on the business logic and the actual functional requirements of the project. The chances that the project is able to be completed within the deadline increases, as they have less work to do. However, if there is a non-functional requirement that requires some sort of caching or if the users of the application are complaining about problems that come along (e.g. lots of traffic or battery usage), you will have to implement the caching system afterwards. Which might force them to redesign parts of the system (both frontend and backend) or forces you to do this in an incomplete way, introducing lots of bugs that usually come along in caching systems. Every platform already has some infrastructure designed to allow caching, e.g. Guava and a local lightweight database. Yet there is more to caching than simply storing information, developers should think about what data is eligible for caching and how this can improve their application. They should be able to do this with an efficient method, customized to the needs of the application considering all possible events that might occur and how they might be able to handle this. These ideas and strategies should then be implemented in both the backend and frontend, however the backend implementation could be reduced to an almost non-existing implementation, since it is not merely about the native application itself, but about the data flow between the different systems.

Because of these reasons, we will try to answer the following questions in this article:

- How can a caching strategy efficiently store and fetch data in a native mobile application?
- What life-cycle events should be considered and how can they react to these events?
- How can the cache be synchronized with the external data service (backend)?

Chapter 2

Methodology

First, a literature study will take place about caching in general: what concepts and principles are used? Which steps can be differentiated when interacting with a cache? What are some common techniques? Some of these will later be revisited and possibly simplified, to use them on a mobile device.

At the same time, research can be done defining which events are commonly handled by a mobile native application. However the case study will be performed in the Android platform, it is important to consider all platforms (Android, iOS, Windows) and how each of them handles these events.

After these first two steps, we can think about how the cache could respond to these different events and why and when this would be beneficial. Depending on the needs of the cache, the ultimate result should maintain the caches integrity, especially concerning business data. Possibly critical business data could be lost or even worse: wrong or insufficient business data could later be synchronized to and from the underlying data service.

Parallel with step 3, a literatur study concerning the synchronization between the frontend cache and the underlying data service will be done. What techniques are commonly used? What happens when there is two-way synchronization and there is conflicting data? Although this is partially out of the scope of the frontend developer, it is important that everyone has a clear view of what they can do to facilitate synchronization. The backend can influence frontend performance and the frontend can influence backend performance!

All these concepts and ideas will then be tested in a case study, which will analyze the performance, from an frontend Android application and a backend Java EE application. When talking about performance, this will be mainly about the performance benefits

on the frontend. Performance benefits of caching in the backend is not in the scope of this thesis.

Chapter 3

Basics of caching

A cache can be defined as a (fast) memory structure designed to reduce costs and improve speeds of computer operations. The term caching is widely used in computer science, from the CPU that uses different multi-level caches, to web caches concerning the reuse of already fetched HTTP pages (Fielding, 1999) and all of them are designed to improve performance. Even low-end devices such as microcontrollers can benefit on the usage of a cache (Hruska, 2016). How this memory structure improves the performance is dependent on the use case in which it used. It could keep the results of long operations or use tokens to check whether data is outdated.

3.1 Basic cache performance metrics

The presence of a cache by itself will not cause a performance increase. It is integrated in a software application that uses a cache. The application then checks whether certain data is present in the cache and if not, does everything it needs to do in order to retrieve this data from a third party. In most examples a cache can be represented by a collection of key/value pairs, to which the application requests a value based on a key. When the cache contains the data the application is requesting, it is commonly known as a *cache hit*. Inversely, when the cache does not contain the requested data, it is known as a *cache miss*. The relationship between the total amount of requests and the total amount of hits is the *hit ratio*. $R = h/n$ where R is the hit ratio, h is the total amount of cache hits and n is the amount of cache requests. These terms are, among others, important factors determining the performance of a cache. A common misconception is that a high hit ratio results in better cache performance, yet this is only true considering hit ratios. If the hit ratio were the only factor, the caches "performance" would be proportional to the size of the cache and a cache of 10Tb always yield better performance than a cache of 20Mb (Wulf and McKee, 1995). This statement can easily be proven wrong when considering other factors such as speed,

as the 20Mb cache will need far less time to produce a result than the cache of 10Tb.

3.2 Paging

However caching is of high importance with lots of aspects in IT, caching is what it is today thanks to some specific problems that have received lots of attentions the last decades. Perhaps the most important problem, considering the field of caching, is the problem with *virtual memory* (Denning, 1970). Virtual memory maps virtual memory addresses, provided by an operating system to a program, to actual hardware memory addresses. This way, the operating system can manage its memory resources and decide where to store and retrieve data from programs using virtual memory. This was considered a very elegant way to increase programming productivity, as programmers could use a high-level API to retrieve and store data instead of needing to talk directly to the processor, which would also make them machine dependent.

The operating system can assign priorities to memory addresses and decide whether it should be stored on a fast or slow memory. The memory of an idle process could for example be persisted on slow memory, i.e. hard drive, so that the faster memory, i.e. RAM, can be used for memory of currently active processes. The moment that the idle process wakes up, it can still use the virtual memory pointers, as if the data has never moved. When a process tries to access a piece of data, currently stored on the hard drive, the operating system will produce a *page fault*. This event will cause the *page*, a block of data, containing the desired data on the hard drive to be transfered to somewhere in RAM memory.

The memory transfer due to a page fault is however very expensive as the hard drive is a lot slower than RAM memory and the processor itself. Therefore it is important to have a good paging strategy, an algorithm that decides which pages will be persisted to the hard drive and which pages will remain in RAM memory. The optimal algorithm will only persist the pages that will not be used for the longest period of time.

The paging problem is very similar to finding the optimal caching strategy, RAM memory is the cache, the hard drive is non-cached memory and we are looking for an algorithm such that only the least interesting memory is removed from the cache. The principle ultimately solving the paging problem is therefore one of the fundamental principles not only in pages and virtual memory, but all of caching, this principle is called the *locality principle*.

3.3 Locality principle

3.3.1 What is the locality principle

When people are executing a process, such as preparing dinner, they always tend to gather everything they need before or during the process. If it were a bigger meal the entire meal is divided in a set of subprocesses. When preparing lasagne it could be divided in preparing the tomato sauce, creating each layer using previously prepared ingredients, etc. The principle of dividing a large task in a set of smaller subtasks is called *divide and conquer*. When creating the lasagne layers, we would first collect all supplies needed (tomato sauce, lasagne sheets, etc) and put them somewhere close, e.g. on the counter, once that is done, the layers are assembled. This is more practical and time saving than to continually walk a relatively long distance each time you need the tomato sauce.

Something similar has been discovered with programs by Denning (2005) when analyzing memory usage of multiprogramming systems using virtual memory in 1967. He formalized this idea through the years and called it the *locality principle* also commonly known as *locality of reference*. He found that programs intrinsically use a very small part of data when performing arbitrary algorithms. In 1980 he formally defined a relationship between the processor, executing the instructions, and the data/object it uses called the *distance* and is denoted as $D(x, t)$ where x is the object and t is the current time. The relevance of this object to the processor at a certain time is based on whether the distance is less than a certain threshold:

$$D(x, t) \leq T$$

where T is the threshold and D(x,t) is the distance function. The threshold and the distance are embedded in a, usually 2 dimensional, spatial-temporal coordinate space where the spatial dimension is the memory address of the object and the temporal dimension is the time when it was accessed.

3.3.2 Interpreting spatial-temporal distances

Because the distance function D is operating in a spatial-temporal space, it is based on the definition and interpretation of the two different dimensions. The easiest way to understand how these dimensions can be interpreted is to look at them seperately. Once that is understood, you simply need a function $m : S \times T \rightarrow \Theta$ where S is the set representing the spatial dimension, T is the set representing the temporal dimension and Θ is the set representing the spatial-temporal space.

Temporal distance, the distance in time that can be expressed in several ways:

- time since last reference
- time until next reference
- access time to get the object
- a combination of the above or others

The time since the last reference to object x can be known by storing the references used by the processor and the time when the reference happened in a list. The time until next reference might yield better results when trying to create an effective and efficient cache since it is known when the object will be used again. However this should require a learning agent that can try and calculate when the next reference will happen based on the reference history. The time unit used should be a relative unit such as the execution of an instruction, this is better than an absolute unit such as seconds because it will ignore external factors that shouldn't affect the distance, e.g. other processes running.

Spatial distance, the distance to the location of the object can be expressed in several ways as well:

- the amount of hops in a network required to retrieve the contents
- the memory on which the data is stored, e.g. objects stored in RAM have a smaller spatial distance than objects stored on a hard drive
- the number of memory addresses between the object and the addresses currently being used or pointed to
- a combination of the above or others

Intuitively, one could see a relationship between the spatial and temporal locality. When the spatial distance to an object is greater, the temporal distance usually also increases because it will need to cover more ground to retrieve the object. When the temporal distance is greater the spatial distance will usually increase because of pacification of the object: it will be stored on slower memory due to the lack of interaction. When checking the relevance of objects to the processor are checked, with $D(x,t) \leq T$, only some objects will pass the inequality, the set of these objects is called the *working set*. It are the objects in the working set that would, among others, be candidates for caching.

3.4 CRUD

When interacting with a cache, there are, in general, several operations that each cache is able to do. In general, you will want all operations that allow you to read the contents of a cache and to get from an arbitrary cache state, to any other arbitrary cache state through, possibly composite, operations. These operations are known as CRUD operations and are a main part of RESTful web services (Battle and Benson, 2008). This will allow an easier implementation when using Read Through (or Cache Aside) Caches. *CRUD* is an acronym for Create Read Update Delete and as the name suggests, 4 operations can be distinguished:

- *Create* allows some client to add an object to a data set. When adding an object to a cache you will typically invoke an add-function receiving two arguments: a key and a value. SQL implements this with its INSERT-statements and HTTP integrates this using a POST request.
- *Read* is used to get all information or a subset of information from a data set, in a typical cache this will be integrated as a get-function receiving the key of the value you wish to access. *The key passed to the function is the same key that is passed to the add-function when creating a new cache entry.* SQL implements this with its SELECT-statements, commonly known as queries, and HTTP integrates this using a GET request.
- *Update* will change the contents of an existing object, this is separated from Create as these two might be treated differently, depending on the system in which the data set is embedded. A cache will mostly provide a common API for both Create and Update: a function with a key and a value as its parameters. Yet they might be seperated into separate functions to increase readability or to simplify event triggering. SQL implements this with its UPDATE-statement and HTTP implements this using PUT-requests.
- *Delete* is used when an element should no longer be an element of the data set. In a cache this might happen when an object is outdated, or if the object has also been removed in the data service. You would then call a delete-function passing the key of the object that should be removed. SQL implements this with its DELETE-statements and HTTP integrates this with its DELETE request.

(Battle and Benson, 2008) Note how these operations are defined on the database-level, software application level, and networking level. Also note how an interface of a cache should at least contain functions allowing the user to execute all CRUD-operations, yet they are not limited by them.

Chapter 4

Cache replacement algorithms

Now that we know *what* should be possible with caches, we can ask ourselves the question *how* we could implement all of this, the algorithms implementing a cache system are called *cache replacement algorithms* or *cache replacement policies*. Yet not just any cache replacement algorithm will do, we are trying to find the most optimal algorithm, the one that will provide the best (performance) results by interacting with it using CRUD-operations. Although most algorithms are only based on your interactions with the cache, some algorithms can be run on separate processes/threads that are then responsible for maintaining the cache even without interactions with the user.

4.1 RR

The first cache replacement algorithm that will be discussed is the Random Replacement strategy. Of all the strategies, this one could be considered one of the strategies with the lowest overhead and the easiest implementations (Zhou, 2010). This makes it ideal for certain low-level systems or hardware that cannot afford many processing cycles on cache management.

Read When trying to get a certain object by its key, the algorithm can use any sorted or non-sorted collection. When using an non-sorted collection it would check each element in the collection in $\mathcal{O}(n)$. Note how this defines the upper bound of the function, therefore it could also be in constant time or $\mathcal{O}(\log_2 n)$ when performing a binary search on a sorted list.

Create When adding an object while the cache is not yet full, it could simply be added to the collection, algorithm-independently. When the cache is already full, the algorithm will randomly remove replace an object from the collection by the new object that should be stored. The operation itself is primitive and easy to program, the only

difficulty is creating a reliable random generator, yet this is not in the scope of this algorithm.

Update Updating is similar to *create*, if the key is present it will result in a *read* to get the appropriate object and then changes the object. If the key is not in the cache, it could either *create* object or alarm the user that it is not present.

Delete When deleting an object, based on a key, it can be done by first performing a *read* to know whether the key is present and if so removing it. Yet some collections will do this nevertheless, in that scenario the algorithm could simply delegate to the delete function from the backing collection.

4.2 FIFO

FIFO, acronym for First-In-First-Out, is a cache replacement policy much like a queue. New objects are inserted at one end of the queue, when the cache is full the first object at the other end will be discarded and the new object will be added at the insertion end (Reineke et al., 2007). FIFO is analogous to FCFS, First-Come First-Served, which is used in OS scheduling algorithms. Note that the hit ratio performance of the FIFO replacement strategy is like the hit ratio performance of the RR replacement strategy (Rao, 1978).

Read A read in a FIFO policy results in a queue being scanned in $\Theta(n)$, almost just like RR. The difference between RR and FIFO is that a FIFO cannot be a sorted collection, since its performance is based on a "constantly" defined sorting structure, this means that you cannot just swap two elements in the cache and expect it to work just the same: you have changed the actual state of the cache. Just like RR, a read does not change the state of the cache.

Create When adding a new object to the cache it will, without loss of generality, be placed on the left side of the queue. When the queue is full, the first element on the right will be removed and all other objects will be "moved to the right" (however this is not an actual operation that occurs in an efficient queue implementation), clearing a space on the left, ready to be filled in by the new object.

Update When updating an existing object, based on a key, there will be a read that will then change the object at the index i by the new object. After the object x is updated, x should then be removed from the queue, every element left of x should

then be moved to the right and x is again added to the left side of the queue. Note that all these operations should be executed in a procedural or atomic way.

Delete When deleting an existing object, you scan the queue until you find the index with value x corresponding to the given key. You will then remove x and move all elements left of x to the right. The next object added to the queue will therefore not cause the object (completely) at the right to be removed from the cache.

4.3 LRU

An *LRU*-cache (Least Recently Used Cache) is possibly the most popular and one of the best cache replacement algorithms out there. There are lots of variants on LRU and therefore it can also be seen as a cache replacement algorithm family. The probability of an object being present in a cache, the hit rate $h(n)$, can be approximated by $h(n) \approx 1 - e^{-q(n)t_C}$, where n is the object being requested, $q(n)$ is the popularity of the object (see *Zipf's law*) and t_C being the (unique) root of $\sum_{n=1}^{N}(1 - e^{-q(n)t}) = C$, with N being the maximum amount of objects in the cache and C the cache capacity. This approximation is called the *Che approximation* (Fricker et al., 2012).

The LRU algorithm is all about replacing the least recently used object, in other words, when a new object should be added to a full cache, it will remove the object that hasn't been accessed for the longest period of time. To know when an object has last been accessed, the usual implementation uses a map, let's call it the *key history map* that can get a timestamp (this can be in milliseconds or can be some relative time unit) based on a key. However it could also be implemented by a queue, where the order of the queue determines which objects have, relatively, been accessed most recently.

Read Retrieving an object based on a key, first boils down to retrieving a value from any collection, just like RR. When the object is found it will not immediately return this, yet it will firstly access the *key history map* and update the timestamp to its newest value. This way the algorithm will know, in the future, that this key has just been accessed. If the LRU is implemented using a queue, it will, without loss of generality, remove the object x at index i, (conceptually) move all objects left of i to the right (which is equivalent to incrementing the index j of all objects where $j < i$) and x will then be inserted at the left side of the queue.

Create If the cache is not yet full, the object will be added to the collection and the key history map will be updated with a new timestamp for the specified key. If the

cache is full, it will first find the key with the lowest timestamp, which is the object that hasn't been accessed for the longest period of time (considering an increasing time function). It will then remove this key in the collection and remove the corresponding key from the map, the object will then be added to the collection and the key history map will be updated with a new timestamp for the specified key. When using a queue with a cache that is not yet full, you will add the new object, without loss of generality, to the left side of the queue. When using a queue with a full cache, you will remove the object to the right of the queue, move all remaining objects to the right, and then add the new object to the left of the queue.

Update The algorithm will check if the key is already stored in the collection, if not, this boils down to *creating* (adding) a new object to the collection or queue. When the key is stored in the cache, it will update the object in the collection to the new object and then it will update the timestamp of the corresponding key in the key history map. When using a queue, it will remove the object with the specified key from the queue, move all objects to the left of the object to the right, and then add the new object, without loss of generality, to the left side of the queue.

Delete To remove an object from the cache, the object is simply removed from both the collection and the key history map. When using a queue, it will remove the object x from the queue and move all objects left of x to the right.

(Day, 2012)This is the basic implementation of an LRU cache, yet there are lots of variants. There are algorithms taking a new approach, based on LRU, there are some algorithms that have a highly specialized implementation of LRU, and there are even some implementations that use LRU together with some other cache replacement algorithm to try and get optimal results based on the situation it is in.

4.4 MRU

The next algorithm is very similar to LRU, *MRU* is an acronym for *Most Recently Used* and will evict the most recently accessed objects when adding a new object to an already full cache. LRU and MRU should not be seen as "competitors" of each other, but rather a twin where, most of the time, one of the two can be chosen based on the context. If it makes sense that a recently requested object has a high probability of being requested again in the near future, you can use LRU. If it makes more sense that a recently requested object will not be requested any time soon, you can choose an MRU cache replacement strategy. Even though one would think LRU would be used most of the time, there are abundant scenario's where MRU would make a lot of sense also. Consider a cache containing Facebook-posts, would it work better under

an MRU or an LRU system? To answer this question further research should be done, but MRU makes a very good candidate, considering people tend not to revisit posts they have just visited 10 minutes earlier. If the user were to respond or like the post, he could however receive notifications of new people posting comments and thus LRU could be more useful.

Read Every object in the cache stores an extra bit (or boolean) called the MRU-bit and is used to find out whether the object has recently been visited, let's say that every object in the cache is stored as a tupel (b, x), where b is the MRU-bit and x is the actual object stored. First the algorithm will check if the object is present in the cache and if so, we'll call the index i and it will set b to 1 or true. This is used to indicate that it has just been accessed. Next the algorithm will check if there exists another index where the MRU-bit is not 1, if so the algorithm ends, otherwise it will flip the MRU-bit at every index back to 0 or false, this is called a *global-flip*. This is the same as checking if $\exists i \in I : B(i) = 0$, where I is the collection of all indexes and $B : I \to \{0, 1\}$ is the function that gets the MRU-bit of the object at an index. This can be refactored as $\forall i \in I : B(i) = 1$, so checking for the existence of an index with an MRU-bit at 0 or false is the same as checking whether every MRU-bit is 1 or true.

Create When the cache is not yet full, the next object will be added to the end of the cache and its MRU-bit will directly be set to 1 or true (ensuring it will not be replaced before the next *global-flip*. If the cache is full it will look for an index j, such that $B(j) = 0$ and $\forall i \in I : i < j, B(i) = 0$, so j is the first index with a 0 or false MRU-bit. Note that there will *always* be at least 1 index with a 0 or false MRU-bit due to global-flipping.

Update If the object is not yet present in the cache, it will delegate the object to *create*. If it is in the cache, it will replace the object, while keeping it at the same position, and set its MRU-bit to 1 or true. If the MRU-bit was previously 0 or false, it is important to check if a *global-flip* should be executed or not. Yet it is always safe to check for this, even if the MRU-bit hasn't changed (and already was on 1 or true).

Delete To remove an object, you first check whether the object is present in the cache, if it is not then you don't have to do anything, if it is present at index i, you move the object at every index j such that $j > i$ to the left which is the same as decrementing j by one, so $j \leftarrow j - 1$.

(Guan et al., 2014) Now that we have discussed the basic MRU and LRU algorithms we can respond to temporal localities with different characteristics.

4.5 LFU

LFU, acronym for Least Frequently Used, is a cache replacement algorithm that looks at how many times an object is accessed. This means that the object that has been accessed the least will be evicted when adding a new object to an already full cache. Just like MRU and LRU, this cache is only benefitial in specific scenario's.

Imagine you are creating a cache for a web server, where static pages, images, css-files, etc. are stored in a cache. This cache will then be accessed with every request to check for its presence, before retrieving the file from some other slower source. One might think LRU is a good idea since the most popular, and therefore the most requested, objects will be present in the cache. This is true to some extent, yet only in some concrete scenario's. Imagine that the "GET /index.html" HTTP request would call n different objects in the cache. If the LRU-cache were to have a maximum size limit N such that $N < n$, the cache will be adding new elements to the cache to later remove it during the same request. This is where LFU might be more meaningfull, even with a maximum size limit $N < n$. The LFU algorithm will give a better ranking to common objects that would be accessed over lots of different request, say a logo or common CSS file.

There are several implementations of an LFU cache, yet I will be discussing the algorithm from Shah and Matani (2010), which ensures $\mathcal{O}(1)$. This implementation uses two data structures, (1) a hashmap where an element can be accessed in $\mathcal{O}(1)$, this is possible if the hash function used is an *injective* function with an upper limit u. It could then map each hash to a single array index of an array of size u which can obviously be stored, retrieved, removed and updated in $\mathcal{O}(1)$. (2) A frequency list, which is a bidirectional linked list that contains an element having a number marking its frequency and having another bidirectional linked list of all cache objects. These cache objects each hold a reference to the frequency element they are in.

Read The hashmap is used to retrieve the cache object based on a key. It will then use its reference to get the frequency element it is in, and use the frequency list to, without loss of generality, traverse to the next element on the right, which should be the next ordered frequency node. If the object were already referenced once, therefore it would be in the 1-node, it would then go to the 2-node. If this node does not exist, it will create it and add it to the right of the current frequency node and to the left of the next frequency node. Now that the cache object is under the next frequency node, the algorithm will check whether the previous frequency node still contains cache objects, if not it will remove the frequency node from the frequency list. Note that however there are particularly more steps than some other algorithms, it is still in $\mathcal{O}(1)$.

Create When the cache is not yet full, it will add the object to the first node of the frequency list, which is the 1-node signifying the object has been referenced once, and will possibly create this node if it is not yet present. If the cache is already full, the first frequency node will be taken (mostly yet not limited to the 1-node) and from there will take any of the linked cache objects and remove if from the linked list and from the hashmap. After this it can just add the object as the cache is no longer full.

Update To update an object, the algorithm will get the cache object, based on its key, from the hashmap and then update the contents of the cache object. After this it will get the frequency node, with frequency number n, based on its reference, go to the $(n+1)$-node (which is possibly created if it does not yet exist), and adds it to its linked list. If the object does not exist in the cache it could either throw an appropriate exception or delegate to the *create* procedure.

Delete To remove an object from the cache, the algorithm will first retrieve the object from the hashmap, based on the key. It will then remove itself from the linked list from the frequency node it is in, possibly removing the frequency node if there are no more elements in it. And last but not least it will remove the object from the hashmap based on the key.

Note how the constant execution time of the algorithm fully depends on the hashmap implementation used. If it is an implementation such as described above, it will provide a constant execution time, yet if it were $\mathcal{O}(\log n)$ or $\mathcal{O}(n)$, the algorithm would no longer have the constant execution time.

4.6 LIRS

LIRS, *Low Inter-reference Recency Set Replacement Policy* (Jiang and Zhang, 2002), is a cache replacement algorithm based on LRU. The major performance problem with LRU is that it always assumes that the object that has been referenced the least recent is the one that will not be accessed for the longest period of time. Yet in some (even trivial) cases, this is simply not the case: imagine a relational database with a B-tree for efficiently retrieving N indexes for a set of N records where there exists a bijective function from an index to a record. In the cache we store blocks of data with a fixed-size where an index is made up of I blocks and a record is made up of R blocks, therefore there are $I \cdot N$ index blocks and $R \cdot N$ record blocks, where $I < R$. Because every operation on a record will first find the record based on the index B-tree, the probability of an index block being accessed will be higher than the probability of a record block being accessed. Yet in a simple LRU-cache it will most of the time favor record blocks in its cache as those are the last referenced blocks in a typical

database operation. A cache favoring index blocks would however be more efficient as it increases the cache hit ratio. *LIRS* tries to solve such a problem by looking at its IRR, *Inter-Reference Recency*, which is the amount of referenced blocks between the last and second-to-last reference of a certain block. Say we have objects A, B, C we could have the following reference history:

$$ABCBABCCBCABC$$

We can see that the initial *IRR* of A is 3 (with BCB in between) and the next is 5 (with $BCCBC$ in between). When adding a new object to the cache A will be the first object to be replaced because it has the highest *IRR*. In a cache implementationt there is a set of objects with a low IRR (*LIR*) and a set of objects with a high IRR (*HIR*), the cache mainly exists of *LIR* objects, which are never considered for replacement. A small subset of the cache's memory is used for *HIR* objects, that are eligible for replacement. When an *HIR* object has a lower IRR than some *LIR* object, the two will be swaped and the *LIR* object becomes an *HIR* object and the *HIR* object becomes an *LIR* object.

An LIRS cache consists of a data structure called a *LIRS stack*, which contains two queues: one used for LIR objects (*LIR stack*), of size L_{lirs} and one used solely for HIR objects (*HIR stack*), of size L_{hirs}. In general $L_{lirs} \gg L_{hirs}$, where L_{hirs} could even be only 1% of L_{lirs}. We say that the object on top of each queue is the most recent accessed object and the bottom object is the least recently accessed object, similar to an *LRU stack*. Stack pruning, which is commonly used in the CRUD operations, is the removal of HIR objects from the bottom of the stack until a LIR object is present at the bottom.

Every object in the LIRS stack is either a LIR or a HIR object, if it is a LIR object, it will always be fully accessible in the LIRS stack and will not be (completely) removed when adding a new object. Every HIR object is either a *resident HIR object* or a *non-resident HIR object*. A HIR object is resident if it is present in the HIR stack, otherwise — if it is only present in the LIR stack — it will be non-resident.

Read When referencing an HIR object it will be added to the top of the LIR stack, removing the bottom element of the corresponding stack and a stack prune will be conducted. If the HIR object was already in the LIR stack it is promoted to an LIR object and its reference in the HIR stack is removed. If the HIR object wasn't yet present in the LIR stack its reference is placed on top of the HIR stack, yet there shouldn't be any object removed from the HIR stack as there is no new item that was added. If the object is a non-resident HIR object, it will still promote to a LIR object, yet nothing will need to be done on the HIR stack as it wasn't present in that stack, by definition. If the object is not yet in the cache, this results in a create operation.

Create When adding a non-existing object to a partially filled LIRS cache, the object will be added to the top of the LIR stack and to the top of the HIR stack. If the LIR stack is already full, the object will be added to the HIR stack, possibly removing the least recent (bottom) object from the stack if it was already full. Note how the removed HIR object could still have a reference in the LIR stack, which would now be a non-resident HIR object.

Update When changing an existing object in the cache it will be the same as a read operation, where the underlying value is changed. It is imperative that this is not simply a delete-create operation, as the replacement policy would then forget this objects history.

Delete When deleting a resident HIR object present in the LIR stack, it is removed from the LIR stack and placed on top of the HIR stack. When deleting a non-resident HIR object, it is simply removed from the LIR stack and will therefore be completely forgotten by the LIRS cache. When deleting a HIR object not present in the LIR stack, the object is removed from the HIR stack and will also be completely forgotten. If the object is a LIR object, it will become a HIR object and will be placed on top of the HIR and LIR stack, possibly removing the bottom element of the HIR stack.

Note how all operations are reduced to operations on queues, therefore the performance of this cache replacement algorithm is directly related to the performance of queues. LIRS is based on IRR and recency, yet neither are explicitely stored or even mentioned in the implementation of the CRUD operations. This is thanks to the carefully defined data structures used in the cache, which have the intrinsic property of implicitly maintaining this information. This not only reduces memory usage, but also allows the algorithm to find the object with the highest IRR (and lowest recency) in $O(1)$.

When removing the bottom element of the LIR stack there will always be a, possibly implicit, stack prune, to maintain the integrity of the stack and make sure that the bottom element of the LIR stack is always a LIR object.

4.7 Bélády's algorithm

Bélády's algorithm is considered the most optimal cache replacement algorithm available, as it discards the objects he knows will not be needed for the longest period of time (Al-Samarraie, nd). Yet there are no claims whether this is the most optimal cache replacement algorithm considering a dynamic cost for retrieving uncached data. Therefore it could be considered the most efficient *page replacement policy* where

every page is a fixed size, yet with the dynamic size of objects in non-page caches this is not the case.

Chapter 5

Conclusion

Curabitur nunc magna, posuere eget, venenatis eu, vehicula ac, velit. Aenean ornare, massa a accumsan pulvinar, quam lorem laoreet purus, eu sodales magna risus molestie lorem. Nunc erat velit, hendrerit quis, malesuada ut, aliquam vitae, wisi. Sed posuere. Suspendisse ipsum arcu, scelerisque nec, aliquam eu, molestie tincidunt, justo. Phasellus iaculis. Sed posuere lorem non ipsum. Pellentesque dapibus. Suspendisse quam libero, laoreet a, tincidunt eget, consequat at, est. Nullam ut lectus non enim consequat facilisis. Mauris leo. Quisque pede ligula, auctor vel, pellentesque vel, posuere id, turpis. Cras ipsum sem, cursus et, facilisis ut, tempus euismod, quam. Suspendisse tristique dolor eu orci. Mauris mattis. Aenean semper. Vivamus tortor magna, facilisis id, varius mattis, hendrerit in, justo. Integer purus.

Vivamus adipiscing. Curabitur imperdiet tempus turpis. Vivamus sapien dolor, congue venenatis, euismod eget, porta rhoncus, magna. Proin condimentum pretium enim. Fusce fringilla, libero et venenatis facilisis, eros enim cursus arcu, vitae facilisis odio augue vitae orci. Aliquam varius nibh ut odio. Sed condimentum condimentum nunc. Pellentesque eget massa. Pellentesque quis mauris. Donec ut ligula ac pede pulvinar lobortis. Pellentesque euismod. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent elit. Ut laoreet ornare est. Phasellus gravida vulputate nulla. Donec sit amet arcu ut sem tempor malesuada. Praesent hendrerit augue in urna. Proin enim ante, ornare vel, consequat ut, blandit in, justo. Donec felis elit, dignissim sed, sagittis ut, ullamcorper a, nulla. Aenean pharetra vulputate odio.

Quisque enim. Proin velit neque, tristique eu, eleifend eget, vestibulum nec, lacus. Vivamus odio. Duis odio urna, vehicula in, elementum aliquam, aliquet laoreet, tellus. Sed velit. Sed vel mi ac elit aliquet interdum. Etiam sapien neque, convallis et, aliquet vel, auctor non, arcu. Aliquam suscipit aliquam lectus. Proin tincidunt magna sed wisi. Integer blandit lacus ut lorem. Sed luctus justo sed enim.

Morbi malesuada hendrerit dui. Nunc mauris leo, dapibus sit amet, vestibulum et, commodo id, est. Pellentesque purus. Pellentesque tristique, nunc ac pulvinar

adipiscing, justo eros consequat lectus, sit amet posuere lectus neque vel augue. Cras consectetuer libero ac eros. Ut eget massa. Fusce sit amet enim eleifend sem dictum auctor. In eget risus luctus wisi convallis pulvinar. Vivamus sapien risus, tempor in, viverra in, aliquet pellentesque, eros. Aliquam euismod libero a sem.

Nunc velit augue, scelerisque dignissim, lobortis et, aliquam in, risus. In eu eros. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Curabitur vulputate elit viverra augue. Mauris fringilla, tortor sit amet malesuada mollis, sapien mi dapibus odio, ac imperdiet ligula enim eget nisl. Quisque vitae pede a pede aliquet suscipit. Phasellus tellus pede, viverra vestibulum, gravida id, laoreet in, justo. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Integer commodo luctus lectus. Mauris justo. Duis varius eros. Sed quam. Cras lacus eros, rutrum eget, varius quis, convallis iaculis, velit. Mauris imperdiet, metus at tristique venenatis, purus neque pellentesque mauris, a ultrices elit lacus nec tortor. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent malesuada. Nam lacus lectus, auctor sit amet, malesuada vel, elementum eget, metus. Duis neque pede, facilisis eget, egestas elementum, nonummy id, neque.

Bibliography

Al-Samarraie, N. A. (n.d.). And page replacement algorithms: Anomaly cases. Baghdad college of Economics Sciences.

Battle, R. and Benson, E. (2008). Bridging the semantic web and web 2.0 with representational state transfer (rest). *Web Semantics: Science, Services and Agents on the World Wide Web*, 6(1):61–69.

Day, T. (2010-2012). *LRU cache implementation in C++*.

Denning, P. J. (1970). Virtual memory. *ACM Computing Surveys (CSUR)*, 2(3):153–189.

Denning, P. J. (2005). The locality principle. *Communications of the ACM*, 48(7):19–24.

Fielding, R. (1999). Hypertext transfer protocol. Technical report, W3C.

Fricker, C., Robert, P., and Roberts, J. (2012). A versatile and accurate approximation for lru cache performance. In *Proceedings of the 24th International Teletraffic Congress*, page 8. International Teletraffic Congress.

Guan, N., Lv, M., Yi, W., and Yu, G. (2014). Wcet analysis with mru cache: challenging lru for predictability. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s):123.

Hruska, J. (2016). How l1 and l2 cpu caches work, and why they're an essential part of modern chips. http://www.extremetech.com/extreme/188776-how-l1-and-l2-cpu-caches-work-and-why-theyre-an-essential-part-of-modern-chips.

Jiang, S. and Zhang, X. (2002). Lirs: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. *ACM SIGMETRICS Performance Evaluation Review*, 30(1):31–42.

Rao, G. S. (1978). Performance analysis of cache memories. *Journal of the ACM (JACM)*, 25(3):378–395.

Reineke, J., Grund, D., Berg, C., and Wilhelm, R. (2007). Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2):99–122.

Shah, A. K. and Matani, M. D. (2010). An o (1) algorithm for implementing the lfu cache eviction scheme. Technical report, Citeseer.

Wulf, W. A. and McKee, S. A. (1995). Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24.

Zhou, S. (2010). An efficient simulation algorithm for cache of random replacement policy.

List of Figures

List of Tables