Technische Universität
Kaiserslautern
University of Kaiserslautern

Fachbereich Informatik
Department of Computer Science

Project Report

Performance Evaluation of Distributed Systems: 2016

Ali Taha, Anusha Halsnad, Manish Kumar

alisafaataha@gmail.com, anusha.halsnad@gmail.com, 1991manish.kumar@gmail.com

February 15, 2017

Supervisors: Daniel Berger
Matthias Schäfer

Contents

1 Introduction

In today networks, the number of users increased tremendously around the world, each individual owning a smartphone or any other device can access the internet in a seamless way. So that, they can access some popular websites or watch some videos. Based on that, we have approximately 3.5 billion internet users. Nevertheless, it is possible in the near future that the number of the users will be much higher because each person is using at least three devices to access the internet, for example (smartphone, laptop, Xbox live account, etc).

Thus, one the most popular website that is used extensively on the internet is Wikipedia, due to the fact that every user around the world, they are accessing Wikipedia more than five times in a month.This introduces a huge challenge for Wikipedia engineers, in which how they can manage this huge number of requests from all the users toward the servers.

The problem can be illustrated in an intuitive way. Every user is using a web browser client (for example Chrome, Internet Explorer, Mozilla Firefox, etc). And we have a web server, in which the Wikipedia engineers stores all kind of the services that the users searching for. Hence, these web servers are finite in size and capacity. Then, every user who is searching for any kind of information, they will send requests to a specific web server, and then the web server responds to the client request in an appropriate way. But if the number of requests triggered by the users considerably large than the capacity of the web server. This will cause the web server to malfunction. As a result, the users will get no response from the web server.

However, In an effort to solve this problem, Wikipedia engineers replicated their servers in many locations around the globe, so that they can minimize this huge number of requests from the users. Although this technique seems promising but it requires some improvement. Therefore, the substantial improvement that it is applied to their web servers is caching

Thus, the cache is a hardware or a software component that is used to store an amount of data, in order that subsequent requests for the same data could be served (directly from the cache) much faster than others. Additionally, the reason for storing this data in the cache might be the result of an earlier computation, for example (the repetitive usage of specific websites).Specifically, the usage of the cache introduced two important terms. First, Cache hit indicates if the requested data found in the cache. Second, cache miss refers if the requested data not found in the cache. [4]

Although the cache makes our lives easier in terms of storing the data and access them very fast, but which cache policy that Wikipedia should use so that they can optimize the storage of data and enable efficient access to them. The Wikipedia engineers must make their choice concerning the cache policies, Because the number of caches policies are relatively large, for example (FIFO, LRU, PLRU, LIFO, GDSF, etc).In particular, each cache policy has advantages and disadvantages, and the size of the cache must be taken into the account. On one hand, the pros for the cache policies are faster access to a valid cached resource, saving a huge amount of bandwidth, collecting useful statistics on web access. On the other hand, the cons are a slow performance if the data does not exist in the cache, receiving an old copy of the data, some of the data getting lost while it is transmitted. [8]

Nonetheless, through the experiments that we exercised during our project, we considered many caching policies (LRU, FIFO, GDSF, LFUDA, GDS, S2LRU). In meantime, we reached a contentment that (FIFO, LRU) have a very effective throughput (the number of request per seconds) in comparison to other cache policies. Interestingly, another cache policies such GDSF, GDS were very promising regarding high object hit ratio, but they were consuming a massive amount of memory. Therefore, we used filter policies (bloom filter) in the essence to eliminate the memory overhead problem.

2 Background and Related Work

2.1 Classical Cache Policies

The cache policies (also called replacement cache policies), they are typically one of the essential key improvements that paved the way of using the cache more efficiently. They could be defined as algorithms, in which a computer program or embedded hardware could utilize them. Therefore, the main goal of such algorithms is to manage the cache content in an intelligent way, for example when the cache is completely filled up with so many contents. Those algorithms will engage by discarding some of the entries in the cache, based on that, they are creating enough space for the new entries. As a result, we have so many cache policies with different approaches, in order to create enough space for the new entries.[2]

Policy Name
Least Recently Used (LRU)
First In First Out (FIFO)
Most Recently Used (MRU)
Pseudo-LRU Tree (PLRUt)
Random Replacement (RR)
Least Frequently Used (LFU)

Table 2.1: List of policies

As shown in Table 2.1, we considered some classical cache policies, because they are broadly used in order to manage the cache. In particular, we will try to explain each of them, so that we could give a brief look about their function and who they are working, which cache policy is preferred in industry, which cache policy has a higher hit ratio (the ratio of an item found in the cache), and lastly which cache policy has small latency when compared to other policies.[2]

The first cache policy is Least Recently Used (LRU). This policy works by discarding the items in the cache that they are least recently used. More specifically, the general implementation of such policy could be accomplished by leveraging the age bits. Those age bits are accompanied with each entry stored in the cache. Thus, the policy will keep track of those old age bits. When the cache is filled up with the items, the cache policy will engage in order to free space for the new items. The cache policy (LRU), it is already pointing to the old entry in the cache based on the age bit. Therefore, it will discard the least item, and replace it with the new item. Consequently, LRU is very popular cache policy, it has a higher throughput in terms of the request for items that could be serviced per second. Nevertheless, LRU has a low object hit ratio and also it is expensive, in which it keeps track of the old age bits (old entries in the cache).[2]

The second cache policy is First In First Out (FIFO). This policy works by discarding the first item who accessed the queue first without keeping track of any kind of items. As soon as the cache is filled up with the items, the cache policy will engage by removing the last item in the queue (first item who enters the queue). Although FIFO has a low object hit ratio similar to LRU when the size of the cache is (10GB) or less, but FIFO has a higher throughput and also it does not leverage pointers or any kind of track toward old entries, which makes it also preferred like LRU in today cache replacement policies.[2]

The third cache policy is Most Recently Used (MRU). This policy works in contrast to LRU by discarding the items in the cache that they are most recently used. Although it seems unnatural, in which that this cache policy evicting the most recently used items in the cache when the cache filled up, but MRU is very effective in many scenarios. In which, this cache policy attains more hit ratio than LRU in situations when the usage of the older entries is typically higher than the new entries. As a result, MRU it is not so preferred in cases such as when the usage of the new items are relatively prime, and it introduces a memory overhead because it holds for each cached item a copy in the memory.[2]

The fourth cache policy is Pseudo-LRU Tree (PLRUt). This policy works by discarding one of the least recently used items, not the least recently one. This cache policy operates by leveraging the usage of bits, each bit is zero or one, and the number of bits is calculated by using a simple formula, which is the cache blocks subtracted by one. Thus, the bits will be arranged in binary tree form. Specifically, all bits will be initialized with zero so that each of them will be going to point in a specific location in the cache. For the sake of simplicity, if we would assume that our cache contains only four blocks, the number of bits will be three. In this case, the root bit will point to one of their successors and also the two successors will point to two blocks in the cache. Now, if there is an access to an item, and this item does not exist in the cache. Hence, the item will be loaded from the memory, and on the basis of the bits pointers, the item will be stored in the cache block. Thereafter, the bits get flipped, in which they point to the least recently block in the cache. Consequently, in comparison to LRU, PLRU has lower overhead and lower latency but it has high miss ratio.[2]

The fifth cache policy is Random replacement(RR). This policy works by discarding the items randomly from the cache. Whenever the case may be that the cache is filled with items. The cache policy will discard any item randomly from the cache, in order, it can create space for the new item to be stored. As a result, this policy with their lightweight function, it has been used in some processors and co-processors for example ARM- processor.[2]

The last cache policy is Least Frequently Used (LFU). This policy works by removing the items, in which their access counts are relatively small when compared with other items in the cache. In case, that the cache is completely filled up with items. Then, as soon as the access count for each item in the cache is obtained, the cache policy will evict the least item that is associated with least access count from the cache, and it is replaced by the new item. Although, LFU seems more or less similar to LRU, but there is an issue regarding the counting procedure. The problem is when an item that is used extensively during a short period of time, which cause that the access count of this item is increased to a large number. Thereafter, this item maybe will never be used in the current future. Nevertheless, this item will not be removed from the cache because of it is own large access count. Which entail that other blocks in the cache even if they are new items, but their access count is small. Thus they will be subject to be removed from the cache.

2.2 Statistics for a Week of Wikipedia Requests

Before going into any further details, we would like first and foremost to give a simple description of the Zipf's law, because we will leverage this concept in the subsequent paragraphs. Following that, Zipf's law is "is an empirical law formulated using mathematical statistics that refers to the fact that many types of data studied in the physical and social sciences can be approximated with a Zipfian distribution". In other words, Zipf's law is used to count the number of objects occurrences, such that those objects counts could be represented in a graph, in order to see which objects that are used more than the others. Interestingly, the Zipf's law is always a good technique to visualize the human behaviors, for example, the tendency of using specific websites.[3]

As shown in Figure 2.1, the x-axis represents the object popularity, and the y-axis represents the number of requests for such objects. What we did in this experiment, we analyzed seven trace files which correspond to seven days. First, we parsed one trace file and we stored the results in an unordered_map. Thereafter, we populated the triple <hashID, count, size> into a vector and sort the contents of the vector from the most popular object to the least popular object. Hence, we repeated the same process with the remaining files. Therefore, the Popularity graph is drawn using request count from the result vs the popularity of the object. Furthermore, the best method to see the results of our popularity graph is to use an example. So, the first popular object (x-axis = 1) has occurred approximately(y-axix= 2098000). The second popular object (x-axis= 2) is occurred approximately (y-axis = 2097152) and so on. Consequently, as we stated earlier, that this popularity graph is depicting the human behavior. In one hand, it is depicting the most popular objects that are accessed extensively (for example, Facebook). On the other hand, it is depicting the least popular objects that are accessed only once.

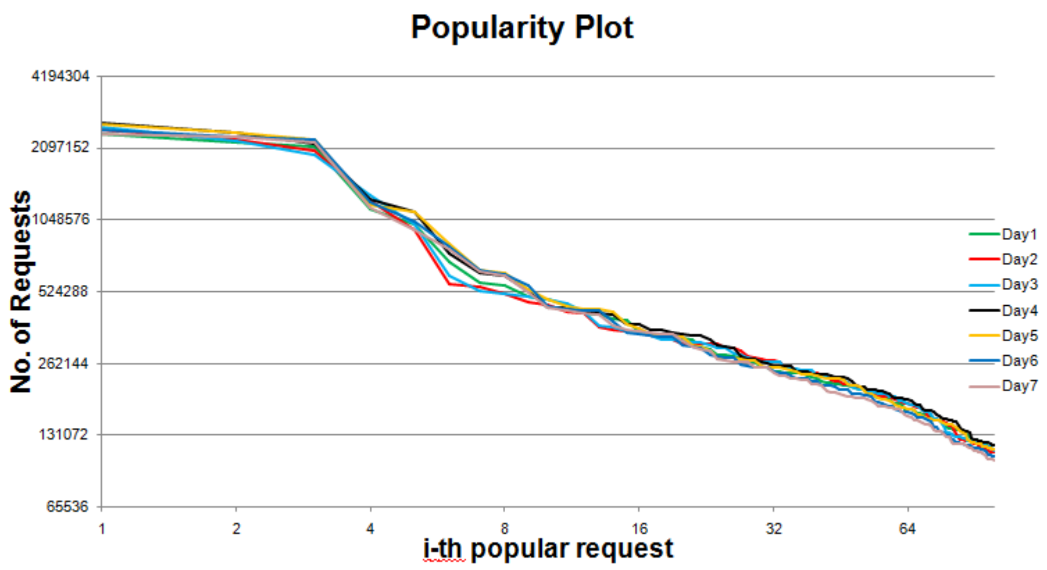

Figure 2.1: Popularity Plot for 7 days

As shown in the Figure 2.2, the x-axis depicts the size of the objects in bytes,(which in this case some of them are multimedia websites). The y-axis represents the number of occurrences of such object size. In this experiment, we utilized the concept of the histogram. Thus, the histogram could be defined as a historical representation of numerical data. In which it can be used to indicate the differences between the data over a period of time, for example, a week in our case. However, what we can see in the graphs can be simplified with examples. in the first day, the object size 512 (in the x-axis) occurred roughly 24 times. In the second day, nearly the same. In the sixth day, the object size (512) occurred 25 times, and in the last day occurred roughly 26 times. As a result, the histogram graphs along the seven days allows us to look more closely at the difference between the data.

Before diving into the next section. We would like to introduce a new technique, which we will employ in the next two paragraphs. It is called cumulative distribution function (CDF). "CDF of a real-valued random variable X, evaluated at x, is the probability that X will take a value less than or equal to x". In other words, CDF will consider first a reference value (in our case an object), thus it accumulates the occurrences of objects from the first object to the referenced object, then it will represent the probability of the occurrences less than or equal to the referenced object.websites.[5]

As shown in Figure 2.3, the x-axis is the size of objects in bytes, and the y-axis is the probability frequency that is confined between zero and one. Hence, we conducted an experiment by utilizing the CDF through the seven days. Although, the concept behind CDF is really complex and not easily understood by the majority of people.

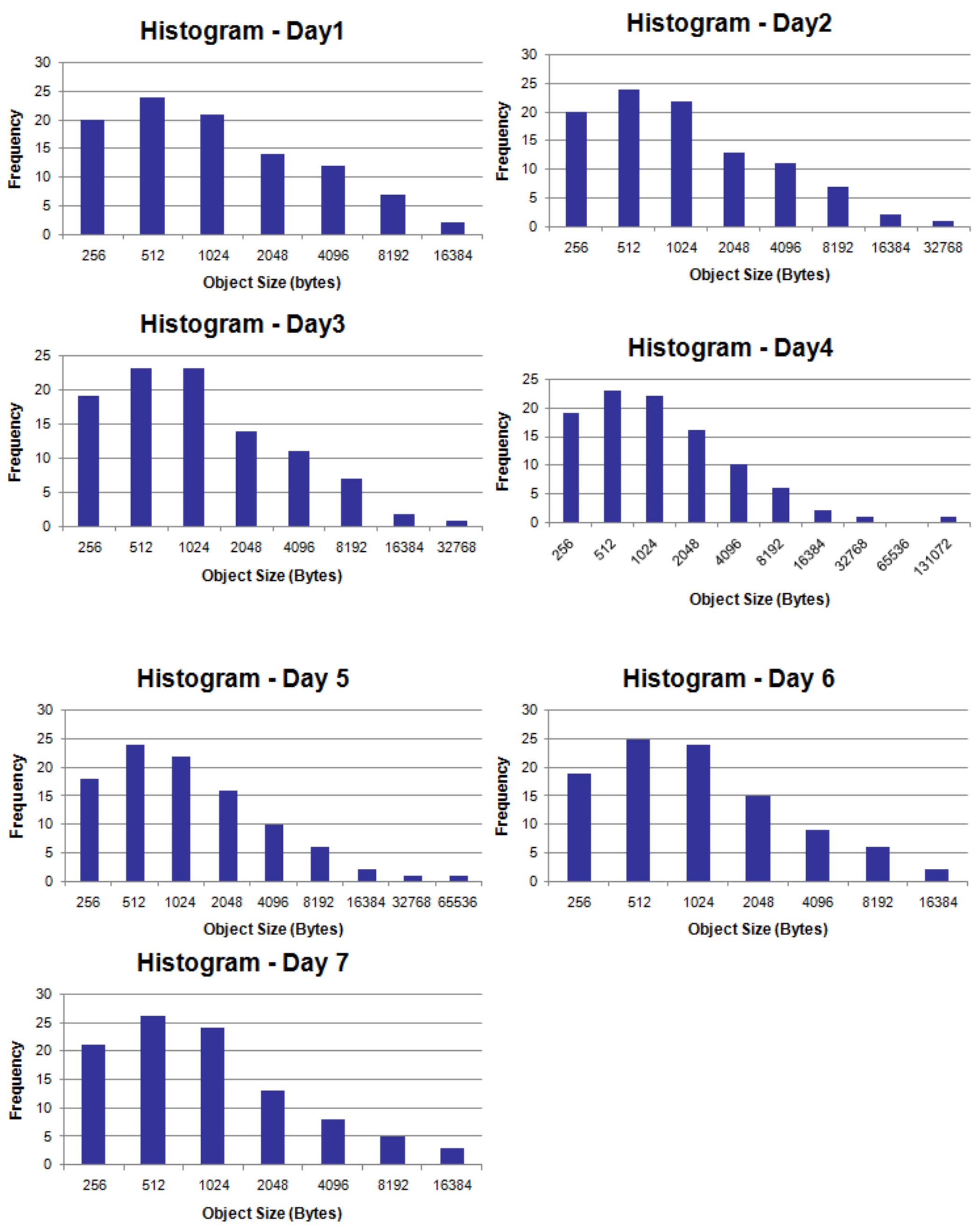

Figure 2.2: Object Size Histogram

But as usual, we will try to introduce an example, which we believe, it will simplify the concept little bit. Thus, we could take the object size 256 (in the x-axis) as a reference value through the seven days, which leads to a probability of 0.2 (in the

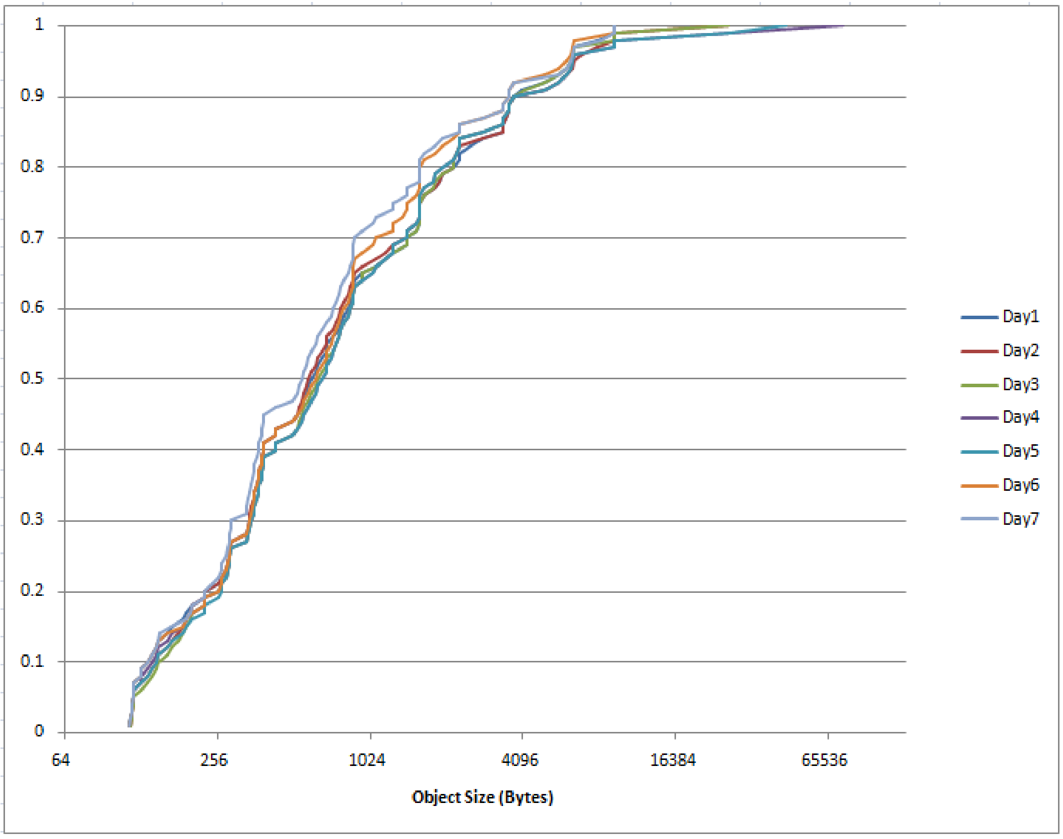

Figure 2.3: Cummulative Distribution Function

y-axis). Therefore the result indicates that the cache is populated with object sizes from 64 till 256 with 20 percent. This process could also be repeated with other reference values (or objects in our case).

As shown in Figure 2.4, , in the x-axis is the number of objects, and in the y-axis is the number of occurrences of the objects. In this experiment, we are revealing the one-hit-wonder problem. It is one of the most common problems in the earlier years because the cache is storing so many objects that are only requested once, which entail that our caches are populated with these objects infinitely often. Thus, it affects the cache efficiency because large space is allocated for such not worthy objects. However, the graph showing us, the huge size of the unworthy objects that is kept in the cache. Based on that, one solution for such a problem is by leveraging filter concept, in which it will get rid off of those objects.

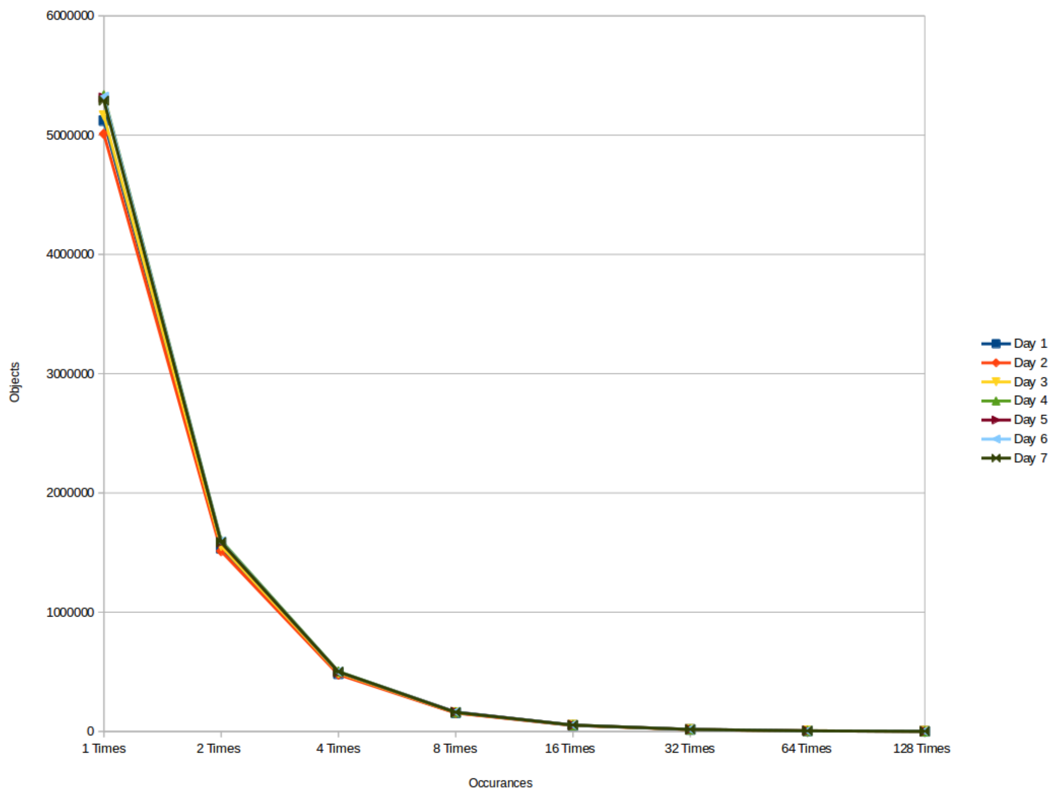

Figure 2.4: n-hit wonders

3 Main Results

3.1 Experimental Setup

Content delivery network (CDN) “is a system of distributed servers (network) that deliver web pages and other Web content to a user based on the geographic locations of the user” [9]. In this section, we will explore the crucial metrics that are used in order to estimate the content delivery network (CDN) performance, for example, object hit ratio (OHR), byte hit ratio,(BHR), throughput, memory overhead. Furthermore, we will give a short description about the simulator that we have employed in our experiments. Afterward, we will list the hardware details (including the operating system (OS), the cache size, random access memory (RAM)).

First of all, we will explain in the beginning the object hit ratio (OHR). OHR is the number of access requests that get serviced in our cache, thus the result of this process yield a ratio, which is interpreted to the number of hits found in the cache. Matter of fact, Instead of sending requests to very far servers (if we don’t use cache), in order to receive any kind of service from those servers. So that, this procedure will consume a huge amount of bandwidth. Thus, the caching of objects has massive benefits (reduce bandwidth, fast access to those objects). As a result, the client and the service provider (SP) will implement the concept of caching both on the client side and service provider side, thus it increases the efficiency and enhances the speed of accessing the resources. Furthermore, the larger the cache is used, the higher the objects that will be stored in it, thus the high probability of attaining object hits. [1]

Second, the byte hit ratio (BHR) is the number of bytes serviced by the cache divided by the overall number of bytes that are requested by the clients. Although BHR seems very similar to the OHR, but it has a substantial difference, and it can be seen clearly when our cache is storing only large objects. So that, the number of objects that is serviced in our cache will be few, which entail definitely less number of hits. Consequently, the larger the objects that are stored in our cache, the fewer the objects that we can keep in it. And it is worthy to note something here, that the average size of the objects on the internet is exactly not known. [1]

Third, the throughput is the number of request per seconds that is performed by the cache. In order to appreciate the merits of using the cache in our systems. We would like to consider an example here, the cache enables us to achieve higher throughput from the underlying resources, in which the processor combines several requests together, thus forming a bundle of requests which can be transmitted only once in a very efficient way from the processor to the cache. Interestingly, the ability to form this bundle of requests will reduce the amount of the bandwidth that is required to transmit the requests in an independent form. However, we should point an extra caution here. If this huge number of request that is triggered by the processor results in a very large amount of miss ratio. It will cause some issues, such as huge request forwarded to the main memory and this will introduce some degradation in the quality of experience. It is recommended to use large caches, in order to avoid a high miss ratio. [1]

Last but not least, the memory overhead is a number of resources that is required in order to implement an operation, in which those resources from one side look like irrelevant but from another side, they seem significant. In particular, every classical cache policy that we already illustrated in earlier sections. They have a considerable amount of memory overhead, which it depends on a number of entries in the cache, for example, filter LRU keeps a copy in the memory for each object in the cache. according to that, filter LRU has a higher memory consumption. Thus, the memory overhead is one of the key factors that we should keep in mind when we are implementing our cache policies.

However, during our experiments, we used a simulator, which is called we (webcachesim). This simulator "simulate a variety of existing caching policies by replaying request traces, and use this framework as a basis to experiment with new ones" [7]. The basic interface of this simulator: ./webcachesim warmUp cacheType log2CacheSize cacheParams traceFiles. The meaning of each part of this interface will be explained as follows, traceFiles are the trace requests, and "the warmUp is the number of requests to skip before gathering statistics", cache type is the cache policy type, log2CacheSize is "the maximum cache capacity in bytes in logarithmic form (base 2)". cacheParams is used to manipulate the cache policies. [7]

In addition, we implemented our experiments regarding the simulation in one system. In particular, we used a Linux operating system (Ubuntu), and the operating system type is 64 bit. The processor is Core i7, 2.40GHz*4. The random access memory (RAM) size is 7.7 GiB.

3.2 Performance Evaluation of Classical Caching Policies

In this section, we will discuss the performance aspects of the classical cache policies. Due to the fact that there is a lot of cache policies, thus we selected some of them, such as (LFU, LRU, FIFO, GDSF, LFU-DA, GDS, S2LRU). Next, we will consider the performance trade-off between the multiple cache policies that are listed previously, for example (memory consumption, throughput, object hit ratio (OHR), byte hit ratio(BHR)). Afterward, we will give some recommendation concerning the optimal choice of the cache policies, with the respect of the cache size.

As shown in Figure 3.1, the x-axis is the cache size (note: the line graph contains many cache sizes 1GB, 10GB, 30GB, 50GB, 100GB, 1000GB, but we will concentrate

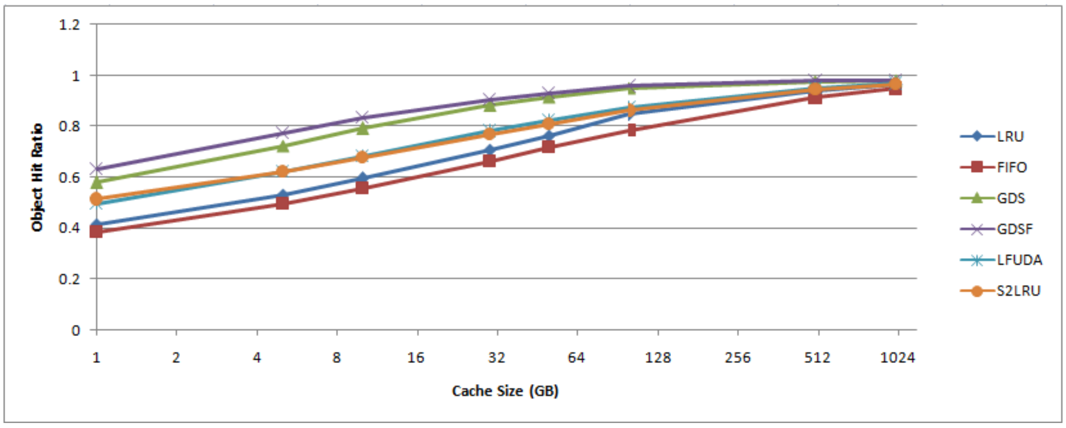

Figure 3.1: Object Hit Ratio Vs Cache Size

only on the 10GB cache in this paragraph and the next one). And the y-axis is the object hit ratio. We employed 10GB cache size in this experiment, then what the figure depicts in this trial that the object hit ratio for GDSF is outstandingly better from other policies. In addition, we can notice also that FIFO and LRU have a worse object hit ratio in comparison to others. As a result, the GDSF and GDS are very effective when the cache size is considerably small, because they are storing valuable objects in the cache by leveraging intelligent algorithm. Which entail, that the hit ratio is extremely great.

As shown in Figure 3.2, the x-axis is the cache size ((the line graph contains many

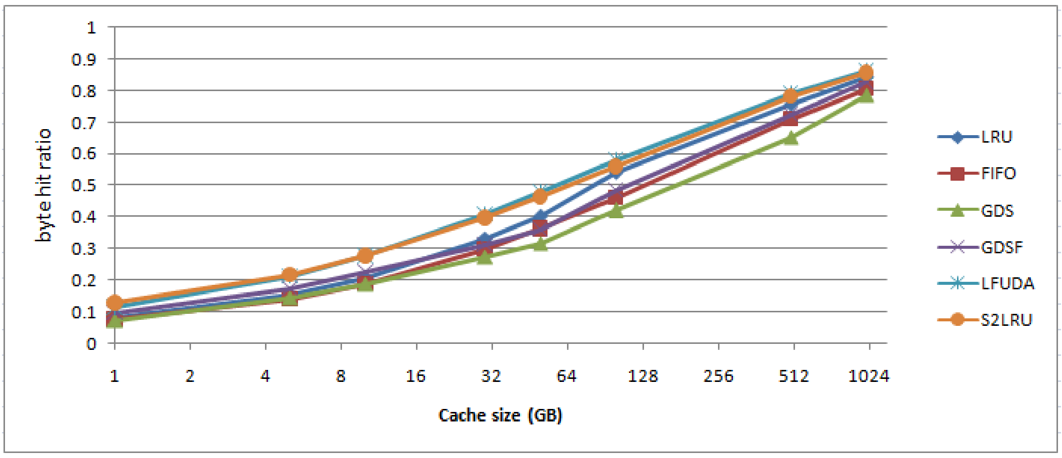

Figure 3.2: Byte Hit Ratio Vs Cache Size

cache sizes 1GB, 10GB, 30GB, 50GB, 100GB, 1000GB, but we will concentrate only on the 10GB cache in this paragraph and the next one), and the y-axis is the byte hit ratio. As previously stated we will focus on the 10GB cache size in this experiment, and what we can grasp from this figure, is that the S2LRU and LFUDA have a low byte hit ratio (in between 0.2 and 0.3), but they are better slightly from all other cache policies. Also, we can notice something that GDSF has a good byte hit ratio unlike others (for example GDS, FIFO). Thus, what we can learn from this experiment, that some of the large objects are cached and the others are not, otherwise we will run out of space very fast.

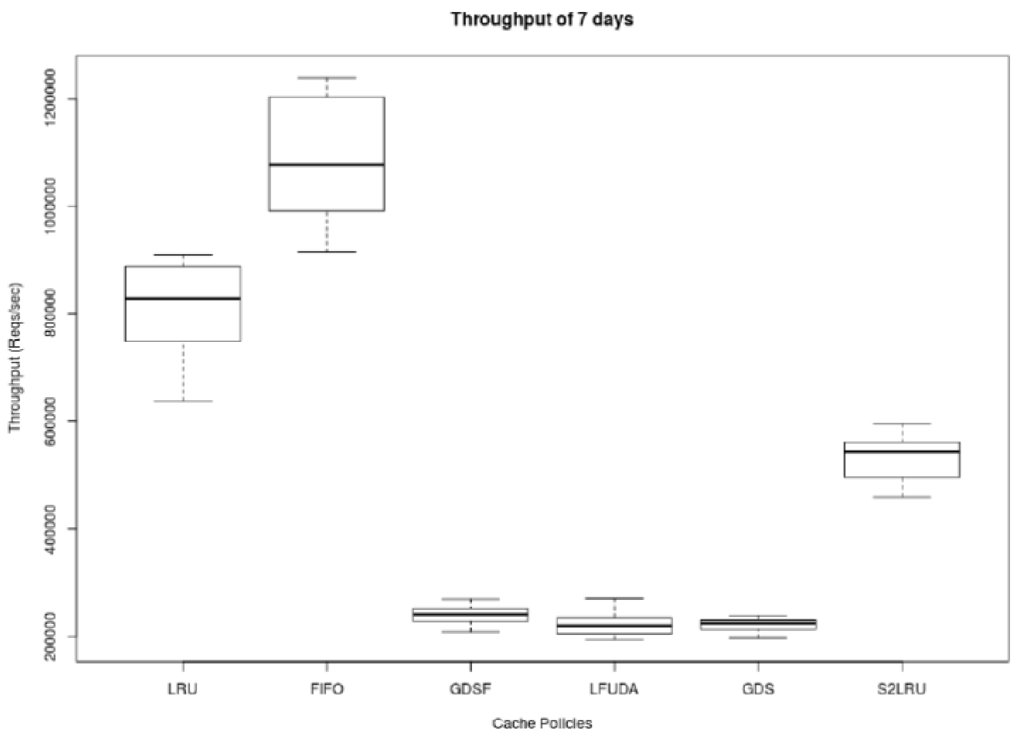

Figure 3.3: Throughput of 7 days

As shown in Figure 3.3, the x-axis is the cache policies and the y-axis is the throughput for each cache policy. In this experiment we took the benefits of the box plot, in order, we can represent the results more clearly, and we utilized a 10 GB cache size. What we can see in this figure that LRU and FIFO have superior throughput, in contrast, GDSF and GDS have very low throughput. Also, we noticed that S2LRU has an interesting throughput, which we will find in a later experiment nearly the same result for S2LRU when we are employing it in a large cache size (1000GB). Therefore, this experiment showed us, that LRU and FIFO have a dominate throughput.

As shown in Figure 3.4, the x-axis is the classical cache policies and the y-axis

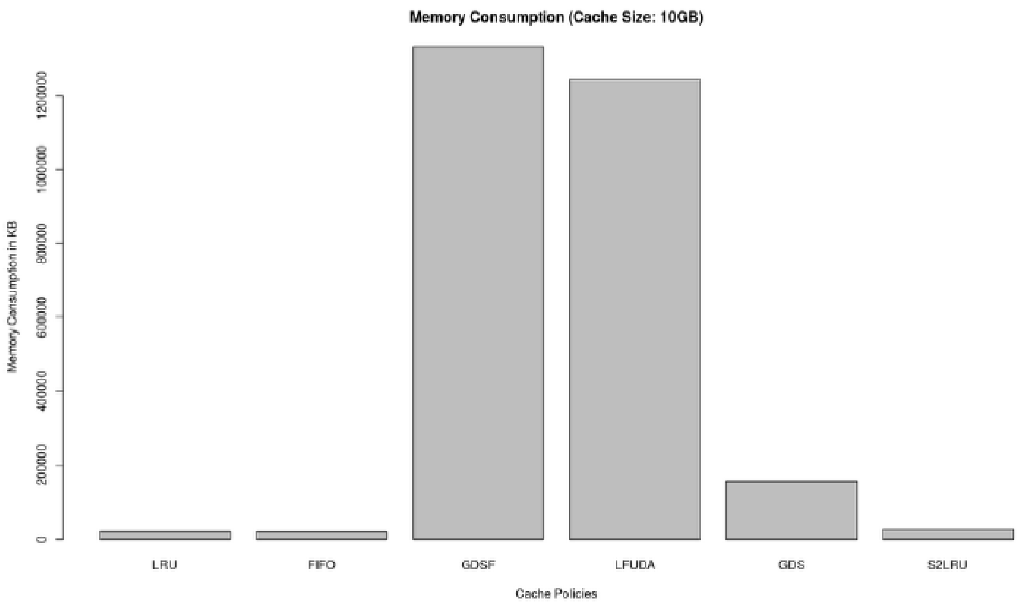

Figure 3.4: Memory Consumption

is the memory consumption in kilobytes (KB) for each cache policy. Through this experiment, we used a 10GB cache size, so that we can distinguish memory consumption for each cache policy. Interestingly, what we discovered in this trial was something noteworthy. The GDSF and LFUDA both have a high memory consumption, and in contrast, we recognized that LRU and FIFO have a very low memory consumption. To this end, what we learned from this experiment, that we should take extra caution regarding memory consumption when we are planning to use both GDSF and LFUDA.

Consequently, what we realized by using 10GB cache size in the last three experiments, that we should find a compromise between the cache policies so that we can choose successfully one cache policy. On the light of the achieved results, we showed that some cache policies have a wonderful object hit ratio while other cache policies have a splendid throughput. In this specific case, the choice of the cache policy will be dependent on the customer preferences. On one hand, maybe the customer will prefer to use GDSF due to the fact that it has a high object hit ratio, which entails that huge amount of bandwidth going through the internet will be reduced, because of the high object hit ratio that is provided by this cache policy. On the other hand, the customer who will choose FIFO or LRU, because it is offering a high throughput. Thus, FIFO and LRU can be used in the data center infrastructure because there they have a massive amount of object requests. For that, they can implement FIFO or LRU to serve this massive object requests.

Furthermore, what we have noticed in the last experiments, that memory consumption has a decisive impact on the cache policy selection. Now, in case we would like to choose GDSF because it has high object hit ratio, but we should keep in mind the huge memory consumption that is caused by this cache policy. Conversely, if we decided to select FIFO or LRU, due to the fact that they have a remarkable throughput, and they have a low memory consumption, but we should not ignore the low object hit ratio that is imposed by those cache policies.

As shown in Figure 3.5, the x-axis is the classical cache policies, and the y-axis is

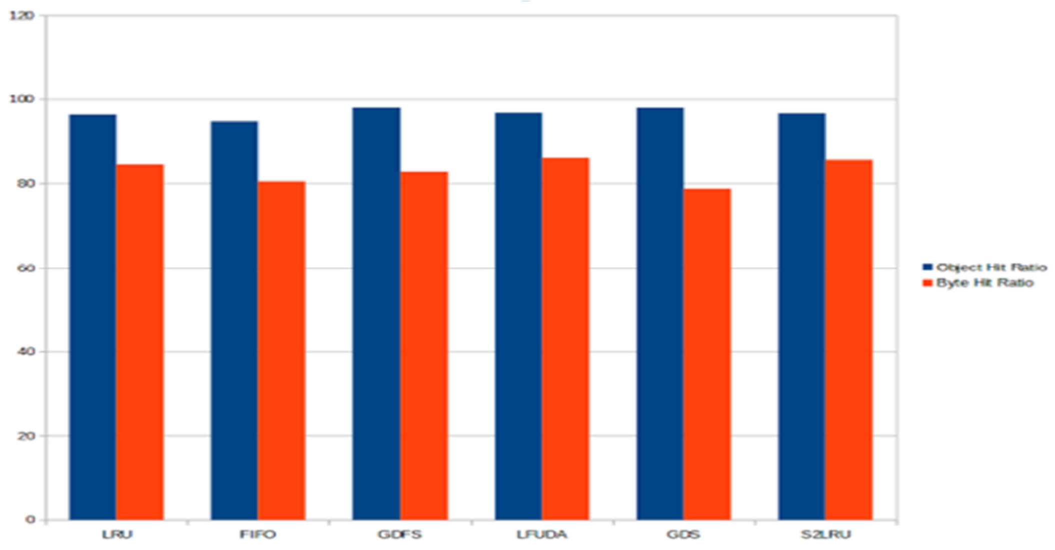

Figure 3.5: Object and Byte Hit Ratio of six Policies

the percentage of object hit ratio and the byte hit ratio. Also, the object hit ratio bars is in blue color, and the byte hit ratio is in red color (to avoid any confusion). we leveraged in this experiment a 1000 cache size. Hence, as we can see in the figure, the object hit ratio for all cache policies is nearly the same and also the percentage of OHR for all cache policies is remarkably high. The reason for such results is obvious because the size of the cache is quite large. Which means that we can store as much as possible objects in the cache. So that, the number of requests that result in hits in our cache are relatively large. One of the best recommendation guidelines that are used as a rule of thumb, is to use a higher cache whenever the chance exist.

Furthermore, as shown in the Figure 3.5, we merged both objects hit ratio (blue bar) and the byte hit ratio (red bars). The byte hit ratios in this figure for all the cache policies are considerably good for example, GDS has roughly 70% byte hit ratio which is the lowest byte hit ratio when compared to others. Thus, the cache size plays an important role, when we are trying to attain high hits in the cache.

As shown in Figure 3.6, the x-axis is the classical cache policies, and the y-axis is the throughput for each cache policy (request per second). In this experiment, we

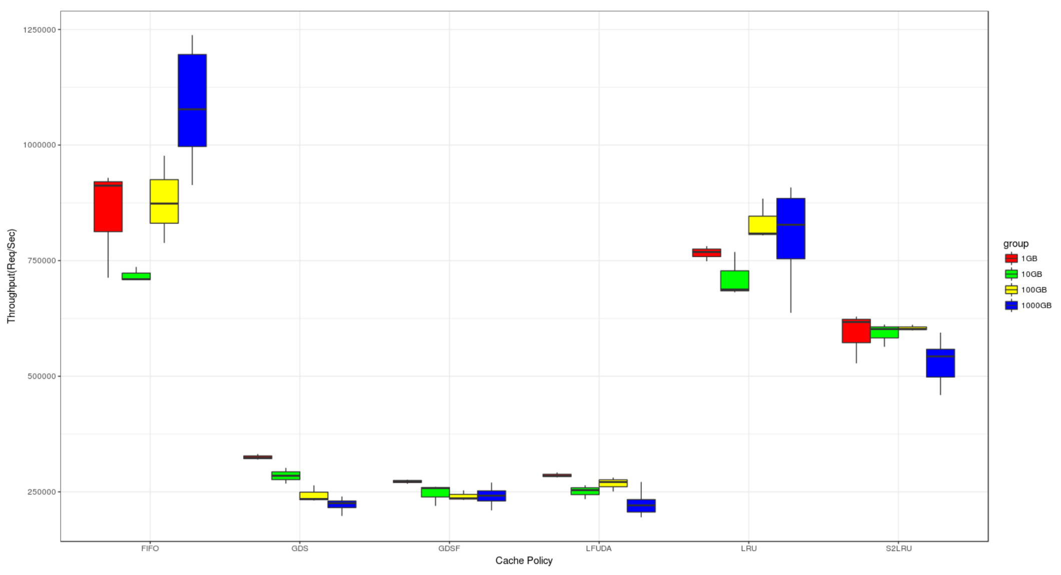

Figure 3.6: Throughput of six policies for 1-1000GB

executed each of the cache policies multiple rounds, in order, we can calculate the minimum, maximum, median throughput for each of the cache policy for the whole seven days. To appreciate the beauty of box plot, we would like to give an example, FIFO has a minimum throughput of 90000 roughly, and the median throughput for FIFO approximately is 1100000, and the maximum throughput roughly is 1200000. However, what we can perceive in this figure is astonishing, in which FIFO and LRU have remarkably a fantastic throughput when compared to other cache policies, in particular, GDSF, LFUDA, GDS they have a very low throughput. Consequently, FIFO and LRU are really useful in terms of throughput, but we must take extra caution when the cache size is smaller (for example 10GB). Because LRU and FIFO have a very low object hit ratio.

As shown in Figure 3.1 and Figure 3.2, the x-axis is the cache size from (1-1000GB) and the y-axis is the object hit ratio and the byte hit ratio. In this experiment, we utilized a variation of cache sizes so that we can see the difference between object hit ratio and byte ratio. This will allow us to choose between the cache policies based on the cache size exquisitely. First, if we consider the object hit ratio for cache size that starts from 1GB till 50GB. We can see in the figure, that policies like GDSF and GDS are the best choice and they are dominant. If we consider the same trial for byte hit ratio, we can see that policies like S2LRU and LFUDA are in the top.

Second, if we take into consideration again the object hit ratio for the cache size that begins from 100GB till 500GB. We can look at the figure and we can infer that policies like GDSF and GDS are still dominant in terms of high object hit ratio. On the other hand, if we consider the same trial for byte hit ratio, we can derive that policies like S2LRU, LFUDA, and LRU are the best.

Last but not least, if we take into consideration again the object hit ratio for the cache size begins from 500GB till 1000GB. We can take a look at the figure and we can conclude that all policies could be chosen due to the large size of the cache. On the other hand, if we consider the same trial for byte hit ratio, we can derive that almost all policies could be selected.

3.3 Maximizing the BHR with Cache Filters

In this section, we will introduce the idea of the cache Filter which does not cache objects until the object is requested a certain number of times depending on the value of the parameter N before it is added to the cache. This is introduced to have better performance in terms of higher byte hit ratio, higher throughput, and better memory efficiency. We will discuss in-depth about picking the optimal N parameter. For example, if N=5, the cache Filter will prevent any object that is requested less than five times (inclusive) to be stored in our cache. Afterward, we will discuss Bloom Filter concept, in which we can achieve lower memory overhead.

As discussed earlier, one-hit-wonder is one of the serious issues. It popped up in the earlier years because it was consuming huge size from the caches. For example, in a content delivery network (CDN) server that was serving web traffic for two days. The statistics that was gathered through this period of time as follows, “74% of roughly 400 million objects in the cache were accessed only once and 90% were accessed less than four times” [6]. Thus, we can infer from the previous example, that the cache is underutilized with so many objects that are only accessed once. The cache Filter introduced so that we can free the cache from these unworthy objects and by storing only the valuable objects that are accessed many times.

As shown in Figure 3.7, the x-axis is the simple Filter policy with different N values and the y-axis is the hit ratio (blue bar = object hit ratio, red bar = byte hit ratio) for the simple Filter. In this experiment, we used a 10GB cache size and we employed the Filter with several N values. We learned two things from the results we obtained. First, the object hit ratio for almost all N values is wonderful. For example, if we choose N=2, which is the lowest value in our experiment, the object hit ratio is nearly 61% percent. Thus, from the results obtained we can say that the principle of using the Filter is totally beneficial in terms of object hit ratio even for a very small N.

Second, the byte hit ratio is considerably low for all values of N. In this case, the choice of N has a slight effect in terms of (high or low) byte hit ratio. We notice a significant improvement in the byte hit ratios for Filter with LRU cache compared to the classical LRU cache.

Figure 3.8, represents throughput comparison of different cache policies for a 10GB cache size. The throughput was measured as requests processed per second. Every policy was run multiple times to obtain a range on the throughput and obtain minimum, maximum and median throughput. We can learn from this figure that though the throughput is acceptable for the Filter but it is less than the throughputs

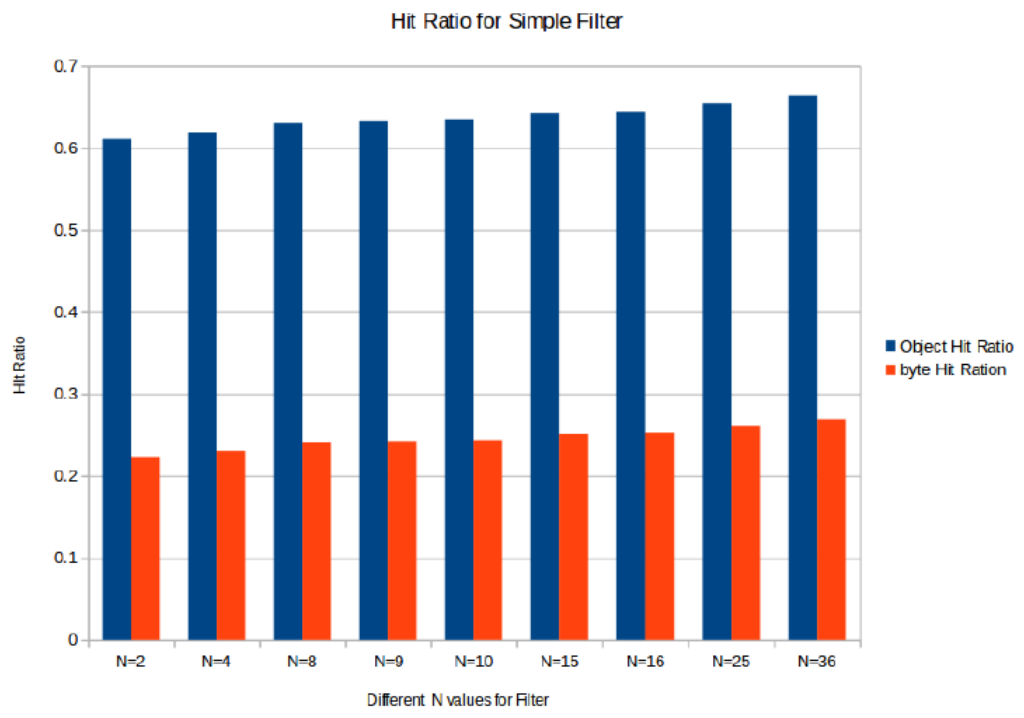

Figure 3.7: Filter Hit Ratios for different N

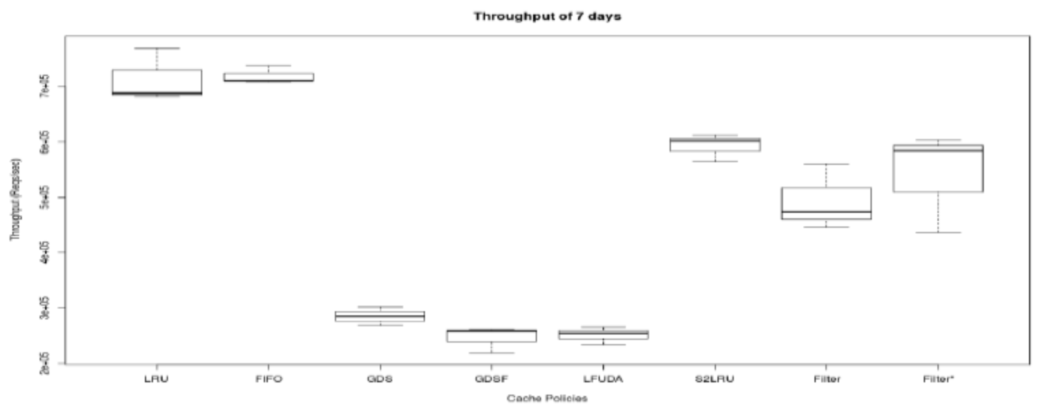

Figure 3.8: Comparision of throughput of Filter with Other Policies

for S2LRU, FIFO, and LRU. Additionally, the cache Filter* with optimal N=25 has slight better throughput than the Filter, but it is still beneath the other cache policies (S2LRU, FIFO, LRU).

Figure 3.9, represents the memory consumption comparison of classical LRU cache

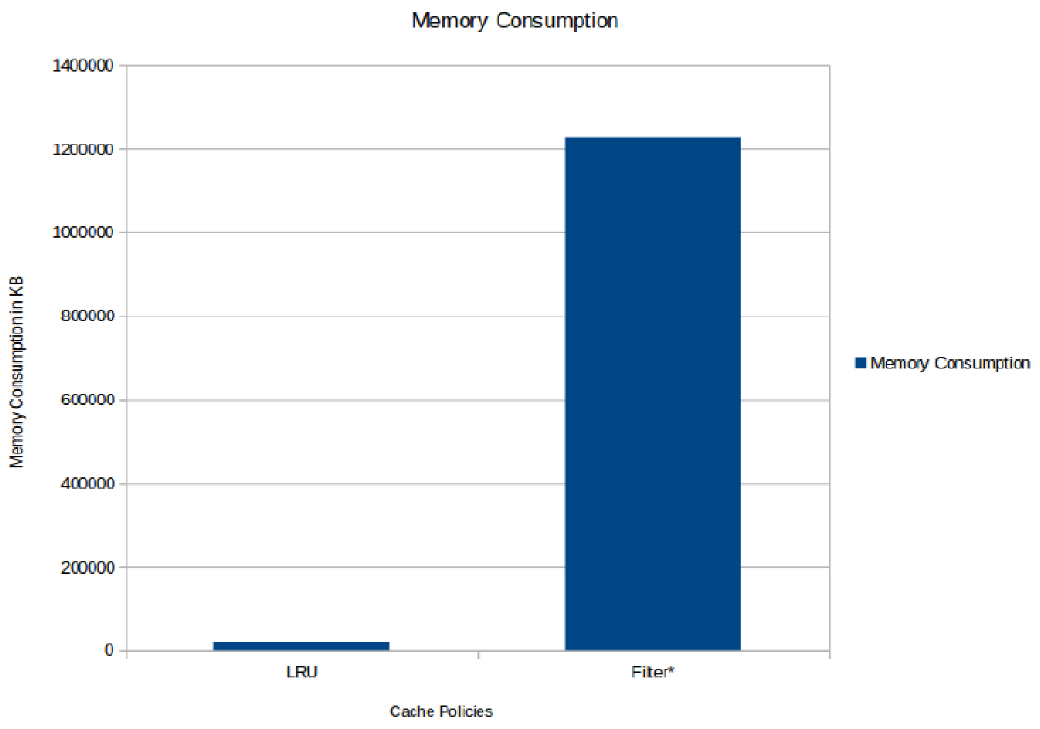

Figure 3.9: Memory Consumption of LRU Vs Filter*

and Filter with LRU. The cache size considered here is 10GB and the memory consumption is measured in kilobytes (KB). Interestingly, the result of this experiment is extraordinary, thus the Filter consumes a massive amount of memory. This means that simple Filter is very bad in terms of memory efficiency. In contrast, we can see in the figure that LRU has very low memory consumption, which is consistent with the earlier experiments.

We learned several things from the experiment. One, the simple Filter showed us a very robust solution for the one-hit wonder problem, thus the cache will be used in a very efficient way. Two, the object hit ratio for the simple Filter is high regardless of the value N, from this we can deduce the crucial significance of the simple Filter for the servers. However, the simple Filter has a disadvantage of high memory consumption, which results in that simple Filter will be used in very rare cases (experimental cases in the laboratory).

Furthermore, we selected multiple N for the simple Filter cache, so that we can gather a variety of results through our trials, which would allow us to choose the optimal N. More specifically, we started with N=10, and the object hit ratio for the simple Filter was totally fine (roughly 61%), and the maximum throughput for the simple Filter was roughly 563803. Then, we increased the value N gradually till we found the optimal value, which was N=25 because the object hit ratio was approximately (65%) and the maximum throughput was roughly 580000. Therefore, based on high object hit ratio and the maximum throughput for the simple Filter, we choose an optimal value for parameter N.

However, the simple Filter might be compared with classical cache policies, so that we can distinguish the relevance of the simple Filter and emphasize the weakness of the simple Filter and the remaining cache policies. To that end, we will examine first which policy is better in terms of the object hit ratio and next, we will consider the byte hit ratio, thereafter we will take into account the minimum and maximum throughput. Then, we will concentrate on the memory consumption for each cache policy. Thus, we can see the differences between the cache policies, which enable us to select the most appropriate one.

Following that, we will start with the object hit ratio. The cache policies like GDSF and GDS have a considerably high object hit ratio (the cache size is 10GB), such that they were dominant over all other cache policies. Additionally, the simple Filter showed us a fantastic object hit ratio regardless of the value N. Therefore, we can conclude that the cache policies such as GDSF and GDS and the simple Filter are very effective and efficient with respect to object hit ratio. On the other hand, cache policies like FIFO and LRU are not recommended to be executed in such instances when the object hit ratio matter because they guarantee a very low object hit ratio. Also, the same could be said about the S2LRU and LFUDA.

Next, we will discuss here the byte hit ratio. The cache policies such as LFUDA and S2LRU are better and preferable cache policies regarding the byte hit ratio. Their results were better than other policies (but it is still poor byte hit ratio) in the experiment that we have conducted. In addition, the simple Filter as we saw earlier, the byte hit ratio was very poor even with the optimal N. Thus, the simple Filter and the cache policies (LFUDA and S2LRU) are not commonly used when the aim is to achieve a high byte hit ratio.

Here, we would like to speak about the throughput. The cache policies, for instance, FIFO and LRU, they were the best and the finest cache policies with respect to the throughput. Additionally, S2LRU has also an observable good throughput, thus it can be used without any hesitation if FIFO and LRU are not available. More interestingly, the simple Filter and the optimal simple Filter (with optimal N), they also have an accepted throughput, but they have not overtaken the throughput which is achieved by the S2LRU. Consequently, FIFO and LRU are predominant in terms of high throughput, and they are recommended if high throughput is the requirement.

Lastly, we would like to examine the memory overhead for the cache policies. Although GDSF has very high object hit ratio which makes it a very attractive solution, it has a huge memory consumption which affects the significance of such a policy. In addition, LFUDA has also massive memory consumption and low throughput, which results in this policy not being a strong candidate for selection. Even though throughput and the object hit ratio of simple Filter and optimal simple Filter are high, they have very high memory consumption. Thus, we should find a better solution, in which we can reduce the memory consumption. However, FIFO and LRU, they are very attractive, because they consume less memory.

To this end, the question that poses itself, how is it possible to achieve a low memory overhead and high throughput Filter in practice. To solve this, we introduce Bloom Filter. From the experiments on simple Filter, we have noticed that it was not a practical approach because of the high memory consumption. Thus, we try to achieve better memory efficiency by maintaining high throughput using the Bloom Filter on LRU cache.

The Bloom Filter idea was introduced by Burton H. Bloom, 1970. The concept behind the Bloom Filter can be explained as follows: the Bloom Filter uses a bit array with finite size (for example, size=2MB) and all bit array entries will be initialized to zero. Then, it will employ a hash function for each object and the number of the hash function is configurable, for instance, the object A is hashed with three hash functions (n=3), and the result of this process will be three array positions (the hash results are limited by the size of the bit array) that they are generated randomly by the hash function. Hence, the three array positions will be incremented by one, which indicates that object A is cached in the Bloom Filter. Now, if there is a request for object A, the three hash function will be computed again to query the object, and if the result (hash function) of all array positions were not zero, the object A is serviced from the cache.

Furthermore, a remark concerning Bloom Filter is that if the queried object is not found in the cache (which mean some of the array positions were zero), the Bloom Filter gives a guarantee that the object doesn't exist in the cache. However, Bloom Filter might introduce false positives for objects that are not in the cache. This happens due to hash collisions of different objects and Bloom Filter cannot guarantee the existence of the object in the cache. For example, if an object is found in the cache with array values set to non-zero it might be due to the increments made by other object insertions and not by the object queried. To solve this problem, more thoughts and advanced techniques are needed.

We implemented Bloom Filter with LRU to realize the ultimate goal of determining a filter cache policy with low memory overhead and high throughput. Therefore, we conducted experiments for different sizes of the array and a different number of hash functions. For the array size, 16MB (l=16MB) and the number of the hash function equals 2 (k=2) we did achieve comparatively smaller byte hit ratio than we achieved for simple Filter. To improve the results, we conducted experiments with greater array sizes. The performance improves as the array size is increased.

Figure 3.10, represents the byte hit ratio versus the array sizes in KB. Through this experiment, we employed different array sizes, to analyze the variation of the results in byte hit ratio. We observe a gradual increase in byte hit ratio for Bloom

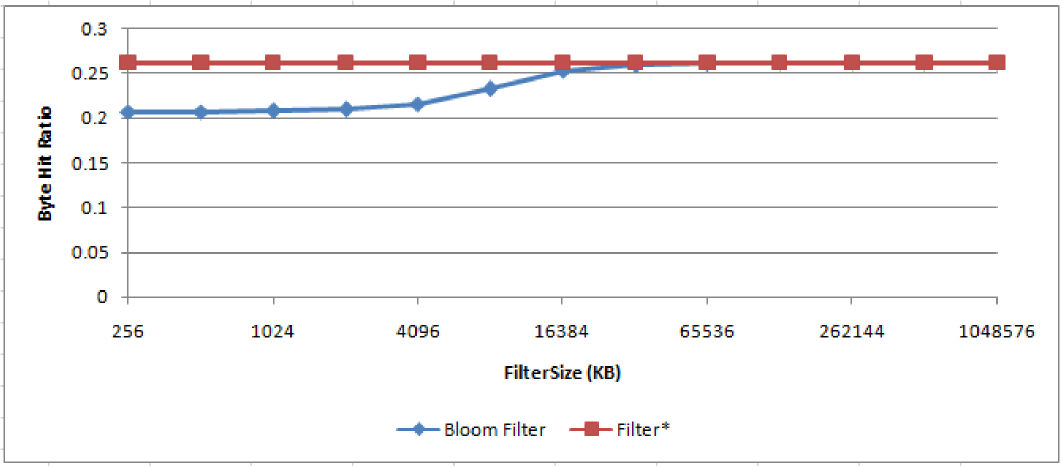

Figure 3.10: Byte Hit Ratio of Bloom Filter in comparision with Filter*

Filter with increasing array size approximately to reaching the Filter* values. We can notice that at 64MB the byte hit ratio of Bloom Filter approximately equals Filter* values. The aim of Bloom Filter was to maintain the hit ratio values achieved in the simple Filter, and we observe that it approximates at 64MB Bloom Filter array.

Figure 3.11, represents memory consumption of Bloom Filter for different array

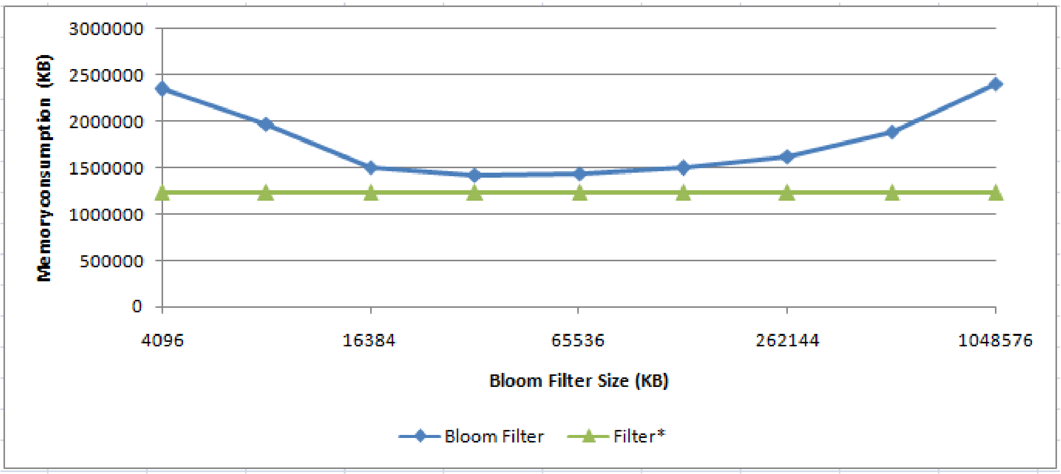

Figure 3.11: Memory Consumption of Bloom Filter in comparision to Filter*

sizes. We wanted to achieve better memory efficiency compared to what we achieved by simple Filter. To this end, we realized some interesting results. Regardless of the optimal array size (in our trial we choose 64MB), the Bloom Filter introduces memory consumption larger than Filter*.

In Figure 3.12, we compare the throughput of Bloom Filter for an array size of

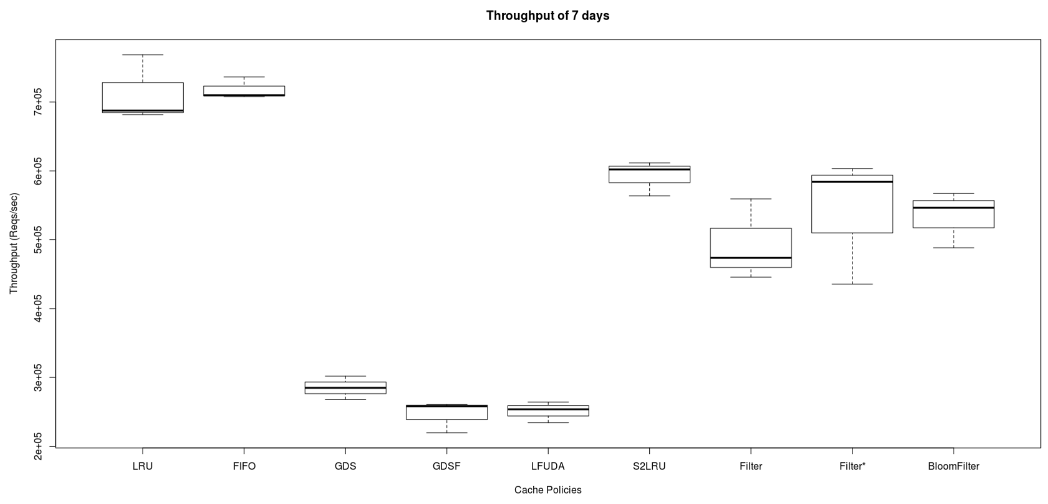

Figure 3.12: Throughput of Bloom Filter in comparision with other policies

64MB with other cache policies we have seen so far. From the figure, we can observe that Bloom Filter yield lower throughput compared to Filter*. Thus, the bloom filter was not capable of enhancing the throughput above the throughput obtained by Filter*.

Therefore, the purpose of implementing the Bloom Filter to achieve better memory efficiency with maintaining the benefits of the improvements we achieved by simple Filter is not served. We gained no improvements over in terms of memory consumption throughput or byte hit ratio by Bloom Filter.

4 Conclusions

To conclude in this report from Figure 2.2, we find out that if object size will increase, the frequency of that object will decrease. As well as we evaluated some classical cache polices (LRU, FIFO, GDS, GDSF, LFUDA, S2LRU) for different cache size (1GB, 10GB, 30GB, 50GB, 100GB, 500GB, 1000GB) and achieved results are available for Object Hit Ration and Byte Hit Ratio in Figure 3.2 and Figure 3.3. We repeated simulation process for each cache policy and achieved the best throughput (Reqs/Sec) for FIFO cache policy, while memory consumption was very high for GDFS and LFUDA cache policy. From Figure 3.5 we are concluding that for higher cache size policies will behave bit different and their throughput will be different in comparison to other lower cache size.

In order to achieve good hit ratio, the idea of the cache filter we evaluated, which will remove some of the objects that are only requested once or twice, and also we can improve the byte hit ratio with cache filter but for best filter we repeated the same simulation with different parameter values and based on Better throughput we picked parameter value 25 for best filter, which is better than some cache policy like GDS, GDSF and LFUDA in throughput. But memory consumption was very high for Filter* in comparison to LRU. Filter* is best cache policy with respect to hit ratio and throughout but with respect to hit ratio and memory consumption Filter* is not good.

To overcome this issue, we implemented "A space-efficient probabilistic data structure: Bloom Filter", as discussed in Section 3.3. We hoped Bloom Filter to provide better memory efficiency that Filter* could not provide. But as we saw in Figure 3.11, we observed worse memory efficiency for our best Bloom Filter, i.e., 64MB Bloom Filter array size. It can be observed from Figure 4.1, that we could not achieve a memory efficient Bloom Filter as we had hoped.

If the Bloom Filter array size is not large enough, we observed huge hash collisions. This allows requests into the cache that were not requested enough number of times. As we have seen, a uniform hash function with large enough array size will solve this problem. But with a wide range of requests, we would require very large Bloom Filter array depending on the number of hash functions involved.

A drawback of this method is that we do not have a mechanism to delete the request count from the Bloom Filter, as we have no record of individual request counts. This will admit new requests without satisfying the threshold. This problem could be solved using Counting Bloom Filter that implements delete operation and restricts the entry into the cache more efficiently.

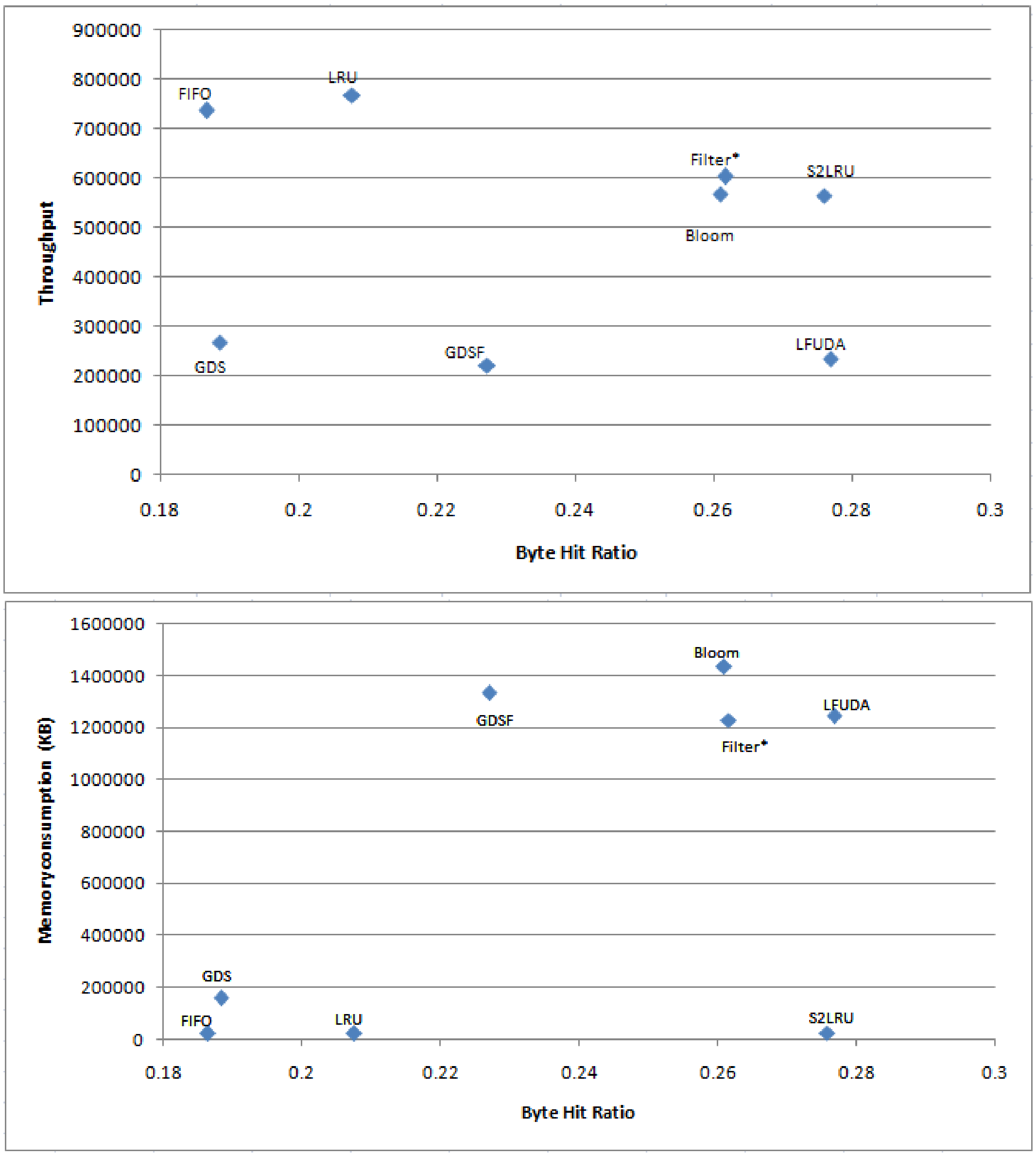

Figure 4.1: Comparision of Policies

Bibliography

[1] Leland R. Beaumont. Calculating web cache hit ratios. 2000.

[2] Wikipedia Contributors. Cache replacement policies. 2016. URL `https://en.wikipedia.org/wiki/Cache_replacement_policies`.

[3] Wikipedia Contributors. Zipf's law. 2016. URL `https://en.wikipedia.org/wiki/Zipf&#39;s_law`.

[4] Wikipedia Contributors. Cache (computing). 2017. URL `https://en.wikipedia.org/wiki/Cache_(computing)`.

[5] Wikipedia Contributors. Cumulative distribution function. 2017. URL `https://en.wikipedia.org/wiki/Cumulative_distribution_function`.

[6] Matthias Schäfer Daniel S. Berger. Performance evaluation of distributed systems task 5. 2016. URL `https://disco.informatik.uni-kl.de/discofiles/teaching/peds17/PEDS%202017-%20Task%205.pdf`.

[7] Mor Harchol-Balter Daniel S. Berger, Ramesh K. Sitaraman. Orchestrating the hot object memory cache in a cdn. 2014. URL `https://github.com/dasebe/webcachesim`.

[8] Scholar Team. 8.1.8 pros and cons of web caching. 2011. URL `http://www.macs.hw.ac.uk/~hamish/3NI/topic172.html`.

[9] Beal Vangie. Cdn - content delivery network. 2010. URL `http://www.webopedia.com/TERM/C/CDN.html`.