CS 6210
Advanced Operating Systems
Project #3
RPC-Based Proxy Server

Abhishek Jain
902985939(ajain333)

April 10, 2015

Contents

1 Abstract

This project aims at developing a simple web 'proxy server' using Apache Thrift, investigating and implementing different caching schemes for the server, and evaluating the performance of the server under different load conditions with different caching schemes.

2 Cache Design Description

2.1 Requirement and Working of a cache

Also referred to as a *Web_proxy_cache*, a function of a proxy server that caches retrieved Web pages on the server's hard disk so that the page can be quickly retrieved by the same or a different user the next time that page is requested. The proxy cache eases bandwidth requirements and reduces delays that are inherent in a heavily trafficked, Internet-connected network. Because the page is stored locally on the proxy server, the page is delivered to the next request at local network speeds. The proxy cache also is advantageous when browsing multiple pages of the same Web site. The proxy cache also stores all of the images and sub-files for the visited pages, so if the user jumps to a new page within the same site that uses, for example, the same images, the proxy cache has them already stored and can load them into the user's browser quicker than having to retrieve them from the Web site server's remote site.

2.2 Data structures used for implementation

All caches that I have implemented are fully associative caches. In order to keep lookup time for these caches minimum, maps were used , which allow for O(1) access. Also other data structures such as queues (e.g. in FIFO) and vectors were used wherever required. All data structures that were used are present in the C++ STL libraries. Multimaps were used wherever required (e.g in MAXS where multiple pages could have the same size). Each cache implementation has atleast one map. Map 1 contains the URL and a data structure that has the data for that URL, and any other entities that the caching policy requires. All policies have atleast one another data structure (queue, vector , map or multimap) which is used to implement the replacement policy.

2.3 Operations the cache supports

All caches support two operations, namely fetch data and update. Fetch data checks map 1 for a hit in the cache. If there is a hit, then the data is returned. However if there is a miss, update is called. If there is space in the cache, new data is inserted in all data structures used by the caching policy. However, if space is less, data structure 2 (and if there is data structure 3 , then data structure 3 as well) is used to find a suitable that can be evicted from the cache. Once it is found, the entry is removed from both the cache and other the data structure as well.

3 Caching Policy Description

3.1 Least Recently Used

3.1.1 Algorithm

In the *Least_Recently_Used* caching policy, the item that has not been requested recently will be removed. This policy expects that the items that were requested recently are more likely to be requested in the near future.

3.1.2 Our implementation

For implementing LRU, 2 maps were used. One map had the URL as the key and a data structure (which contained the data returned by curl for that particular URL along with the timestamp) as the value. The second map had the timestamp as the key and the URL as the value. Both maps were sorted in increasing order. When there was a hit in a cache, the data structure's timestamp for that particular URL is updated in map 1 and in map 2, I first remove the entry for that URL and then reinsert the URL with the updated timestamp in map 2. On a miss, the URL along with the data and the timestamp is inserted in both maps. When the cache is full and new data needs to come in , map 2 is checked for lowest timestamp (i.e the first element of map2), the corresponding URL for this timestamp is obtained, and this URL is deleted from both maps along with the data and the timestamp.

3.1.3 The good

It has good performance over high locality workloads.

3.1.4 The bad

It can have a pathological behaviour for memory intensive workloads that have a working set greater than the available cache size. If a Program that references pages in a cyclic fashion, the least recently used page becomes next page to reference and thus forces LRU makes precisely the wrong choice for replacement.

3.2 Least Frequently Used

3.2.1 Algorithm

In *Least_Frequently_Used*, the item that has been least frequently used in the cache will be the primary candidate for eviction. The idea is that the items that were frequently used in the past will be frequently used in the future.

3.2.2 Our implementation

Again 2 maps were used like LRU. The first map has the URL as the key and the data structure (containing frequency and data) as value for the URL. But this time, in the second map, entries were inserted using the frequency as the key and the URL as the value for that key. Map 2 was again sorted in increasing order , and on eviction, the first entry in map 2 is removed. The URL corresponding to the first entry in map2 is searched in map1 and that complete entry (including the URL and data structure) is removed from map 1.

3.2.3 The good

Efficient and easy to implement algorithm.

3.2.4 The bad

Consider an item in memory which is referenced repeatedly for a short period of time and is not accessed again for an extended period of time. Due to how rapidly it was just accessed its counter has increased drastically even though it will not be used again for a decent amount of time. This leaves other blocks which may actually be used more frequently susceptible to purging simply because they were accessed through a different method.

3.3 Largest Size First (MAXS)

3.3.1 Algorithm

In $Largest\_Size\_First(MAXS)$, the size of an entry in the cache is checked. It will evict the entry with the largest size in the cache. This policy assumes that the documents with a large size are less likely to be accessed because there is a high access delay associated with such documents.

3.3.2 Our implementation

Again 2 maps were used like LRU. The first map has the URL as the key and the data structure (containing size and data) as value for the URL. But this time, in the second map, entries were inserted using the size as the key and the URL as the value for that key. This time Map 2 was sorted in decreasing order , and on eviction, the first entry in map 2 is removed. The URL corresponding to the first entry in map2 is searched in map1 and that complete entry (including the URL and data structure) is removed from map 1.

3.3.3 The good

Evicts bigger pages which frees up space for smaller pages.

3.3.4 The bad

Will be a problem if bigger pages are continuously accessed.Also smaller pages never get evicted even though they are not being used, causing cache pollution.

3.4 Random

3.4.1 Algorithm

In *Random*, as the name suggests,an entry from the cache is evicted at random. It is not an intelligent caching policy and does not perform particularly well under any load. The good part is that this policy does not perform poorly as well as compared to other loads.

3.4.2 Our implementation

Again 2 maps were used like LRU. The first map has the URL as the key and the data structure (containing size and timestamp) as value for the URL. In the second map, entries were inserted using the size as the key and the URL as the value for that key (not required but allowed code reuse). On eviction, a random entry in map 2 is removed. The URL corresponding to this random entry in map2 is searched in map1 and that complete entry (including the URL and data structure) is removed from map 1.

3.5 FIFO

3.5.1 Algorithm

This caching policy works like a queue. The item that was added in the cache at the very beginning will be removed next. This makes sure that pollution by old items will be prevented. This policy keeps track of when the item was added to the cache.

3.5.2 Our implementation

Here a map and a queue were used. The first map has the URL as the key and the data structure (containing size and data) as value for the URL. As soon as a URL, that is not present in the cahce is encounterd, the URL is inserted in the queue. When the time for eviction comes, the first entry of the queue is removed from the queue and the corresponding URL (and its data) is removed from map 1 as well.

3.5.3 The good

It works well for data subject to sequentiality.

3.5.4 The bad

Its not efficient for data subject to locality concerns. Increasing the cache size does not benefit its efficiency.

3.6 LRU-MIN

3.6.1 Algorithm

This policy is a variant of LRU which tries to minimize the number of documents replaced and gives preference to small-size documents to stay in the cache. If an incoming document with size S does not fit in the cache, the policy considers documents whose sizes are no less than S for eviction using the LRU policy. If there is no document with such size, the process is repeated for documents whose sizes are at least S/2, then documents whose sizes are at least S/4, and so on.

3.6.2 Our implementation

Here 3 maps were used. First map acts as an associative cache, second one acts as a sorted (in decreasing order) container that contains URLs sorted by size in decreasing order, and the third map that contains the URLs sorted by timestamps in increasing order. Using these 3 data structures, the LRU-MIN algorithm was implemented as mentioned above.

3.7 Greedy-Dual-Size(GDS)

3.7.1 Algorithm

This caching policy removes item with lowest value of H = (cost of bringing item with to cache/size of item). When an item is replaced, decrement all other items' H value. When items are accessed again, restore their H value to the initial value. The h value is parameterized. For our implementation, I have used H as 1/size of page.

3.7.2 Our implementation

Here just one map is used which acts as the cache. When the time for eviction comes, the entire map is iterated to find the URL with the lowest value of H. Once found, it is removed from the cache. I have used 1/size of page as the value of H for all entries.

3.8 Greedy-Dual-Size-Frequency(GDSF)

3.8.1 Algorithm

The Greedy-Dual-Size-Frequency algorithm tries to improve the existing GDS policy. It does so by incorporating file frequency count frequency in the computation of H.

3.8.2 Our implementation

This again uses 2 maps with map1 acting as a fully associative cache and map2 containing entries which are sorted in an increasing order on the basis of their priority. Map 2 is modeled in such a way that it acts as a priority queue. Using map1 and the priority queue (map 2) the GDSF algorithm is implemented as mentioned in the paper HPL-98-69R1. For our implementation, I have chosen a cost function of 2+(size/536).

4 Metrics used for evaluation

4.1 Hit ratio

The number of requests satisfied from the proxy cache as a percentage of total requests. The "hit ratio" of a cache describes how often a searched-for item is actually found in the cache. It is highly useful when each cache entry is of a constant size.

4.2 Byte Hit ratio

The number of bytes that are transferred from the proxy cache as a percentage of the total number of bytes for all requests. This metric is used to specially evaluate proxy caches as the data size of the entries in the caches are not constant. It expresses the number of bytes that the cache was able to serve with respect to the number of bytes that the client actually requested and thus gives us a better idea of how successful the cache actually was in servicing requests. An important point to note is Hit ratio and Byte hit ratio may or may not give the same trends as the trends observed are highly dependent on the access patterns and the size of the pages being requested.

4.3 Average access Time

Average access time is another metric that can be used to measure the effectiveness of the cache. However, since many factors such as wifi strength, the hardware on which the experiments were being run, web traffic etc could have affected our readings, I have not used this as a performance metric. One thing to note is that as cache size increases, (if all other factors remain constant) the Average access time will decrease.

5 Workloads used for Experiments

5.1 Inspiration behind workloads

The policies that I want to draw the readers attention to are Least Recently Used, Least Frequently Used, Random and Largest Size First, though I have implemented and presented the results of other policies as well. I wanted to generate workloads that would help us present to the reader, the cases where a certain policy would perform better than the other and find out a suitable policy for each workload. With this in mind, two workloads were generated. Workload one brings out the disadvantages of using MAXS (Largest size first) and how LRU/LFU can overcome its short comings. Workload 2 does the opposite. To know about the workloads, please, continue reading.

5.2 Steps for generating the workloads (common stps)

- Search the web for 125 unique URLS and make a list of the same.
- Sort this list of URLS based on the size of the documents that correspond to these URLS in increasing order.
- Use binomial distribution (on this sorted list of URLS) to convert these 125 unique URLS into 1000 accesses (described below) .

5.3 Steps for generating workload 1

- Use binomial distribution to generate 1000 accesses with probability of success on a single trial denoted by p=0.75.
- By using binomial distribution I can generate 1000 accessses. The accesses will have repeated URLS. The range of URLS in the sorted list from which these URLs are selected (to generate an access pattern with repeating accesses) is determined on the basis of the value of the p.
- A value of p=0.75 means that URLs that are present in the last quarter of the sorted URL list will be selected to generate repeating accesses. Remember, that the last quarter of the list contains URLS whose documents have the maximum size. Thus I are generating an access pattern that has URLs whose average document size is pretty large.

- The frequency of these URLs is decided by the binomial curve. The URL present on the position 0.75x125 (ie 94) will have maximum frequency with the frequency of URLS on its side, ie on positions 93 and 95 having frequency less than the URL on position 94 but more than the URLS on position 92 and 96 and so on.

- The C++ random library has a binomial distribution function which is used to generate these accesses. Once the original list of URLS is read and stored in a vector in an increasing order, a pattern of 1000 accesses is generated using the binomial function, which is used to generate 1000 indexes into the vector mentioned above, thus generating 1000 accesses. All these accesses are stored in a file called Workload1.txt which is then fed to the client.

5.4 Steps for generating workload 2

- Most of the steps remain same for this workload as well. The main distinguishing point is that 2 binomial distributions (each with different values of p) are used.

- For the first binomial distribution, p value = 0.5. This allows us to use URLs whose average size is on the smaller side.

- For the second binomial distribution, p value = 0.9. This allows us to use URLs whose average size is pretty high.

- Now a group of 10 accesses is generated. 8 URLS are selected from the group of URLS whose average size is small (which is selected by using a binomial distribution with p =0.5). 2 URLS are selected from the group of URLS whose average size is large (which is selected by using a binomial distribution with p =0.9).

- Once I have this pattern of 10 accesses , a list of 1000 accesses is generated by doing the same thing 100 times, thus generating 100 groups , each having 8 accesses that have small document size and 2 accesses that have very large page size.

5.5 The workload generator

I have also included the workload generator program to generate the workloads mentioned above. It is present in the submission package as Payload-Generator.cpp .

6 Experiment Description

6.1 Experiment Methodology

- The two machines that were used for the purpose of the experimentation were 2 laptops with each of them having an i7 intel quad-core processor with Ubuntu 14.10 installed on them.
- They were networked over LAN with 20GB/s bandwidth.
- Apache thrift 0.9.2 was used with Libcurl library.
- I measured Hit Rate and Byte Hit Ratio.
- Workloads that were generated above were used for carrying out the experiments.
- The client uses the workload files to generate requests.
- The server responds to the requests.
- In order to make the testing transparent and fast, a unique technique was used. The server implements and uses all the different cache types parallelly. This allows us to test all different caches simultaneously. The advantage of this approach is that each cache sees all the accesses of the trace file (as is expected) but the speed with which the tests are carried out improves greatly.
- Each cache type has its own counters which count the number of hits and misses. There is a global counter for timestamp, which is incremented each time an access is requested. The client also has a global data counter which keeps a track of all the bytes requested.
- All the caches send the number of misses to the client one by one once all accesses are completed. Then on the client side, hit ratio is calculated by using the miss count for each cache type.
- Similarly all the caches also send the number of bytes that were found in the respective caches to the client once all accesses are completed. Since the client has a record of the number of bytes that were requested, byte hit ratio for each cache is calculated on the client side.
- On the server side, when server sees a request, it tries to see if any of the caches implemented has the requested data. If one cache does have the requested data, all other caches that do not have the data are

updated by using the cached data. Thus each cache sees the request and works independently.

- If no cache has the data then URL is requested using CURL and all caches are updated.
- This greately increases the speed of tests that will be carried out no matter how large the trace file.
- In order to run the server and client on the same machine, either run the TestPayload1.sh or TestPayload2.sh to run server and client on same machine. This script runs all tests with the requested workloads by changing cache sizes and storing all the generated results in a text file.
- Even though tests work with 2 different machines, the submitted code uses localhost so as to allow running on a single machine without the hassle of setting up different machines.
- The following acache sizes are used : 0(no cache),128KB, 256KB, 512KB, 1024KB, 2048KB, 4096KB, 8192KB.

6.2 Hypothesis and Expected Results

6.2.1 For WorkLoad 1

- If you remember, workload 1 has a pattern where all the accesses consist of pages that are comparatively large in size and the difference in sizes of the pages used in the workload is not that huge.
- In such a scenario, Largest Size first will throw away those documents which are largest in size even though they are being accessed regularly. This would result in repeated misses and thus reduce the hit ratio. A similar thing could also be said for the byte hit ratio.
- In the same scenario, on the other hand, a LRU/LFU cache will try to keep pages that are being accessed frequently , irrespective of their sizes. This should lead to larger hit ratio and byte hit rate.
- For random , as the name suggests, the results are random and cannot be predicted.

Hence, to summarize, I believe that the MAXS cache would perform poorly when compared to LRU/LFU caches, when workload 1 is used. Also I believe that this trend is applicable to both performance metrics. Lastly, I believe that there is a high probability that this trend does not change when cache sizes are changed.

6.2.2 For WorkLoad 2

- If you remember, workload 2 is composed of groups of documents where there are 8 lightweight documents followed by 2 large size documents. The pattern consists of 100 such groups.
- In this scenario, MAXS will remove the document with the largest size when a new document comes in whereas LRU/LFU remove documents based on lowest timestamp or frequency.
- Now suppose the first 8 documents of a group fit in both the MAXs and LRU/LFU caches. The 9th document that comes in fills both caches. Now the 10th document that comes in will have different effects in different caches.
- In the LRU cache, the 10 th document will be the most recently used and thus the 8 small documents stored earlier will be thrown out of the cache in order to accomodate the new large document. Now when the next group (comprising of 8 small and 2 large documents) comes in which will have atleast some of the small documents which were present in the previous group (because I are using binomial distribution) , there will be increased misses. I may have a hit on the large document, but since the number of large sized documents in a group is small, the number of misses will dominate.
- On the other hand, in the MAXS cache, the 10th document of the 1st group will be replace the 9th document of the 1st group since it has the largest size. Now when the second group comes, I see more hits and less misses.
- However the byte hit ratio may or may not follow the exact pattern because the byte hit ratio depends on the number of bytes found in the cache. So if there are more repetitions of pages with larger sizes than repetitions of pages with smaller sizes, LRU may have higher byte hit ratio. On the other hand, if the opposite is true, then MAXS will have better byte hit ratio.
- Results of random , once again cannot be predicted.

Hence, to summarize, I believe that the MAXS cache would give better performance when compared to LRU/LFU caches, when workload 2 is used. I also believe that there is a high probability that this trend does not change when cache sizes are changed.

6.2.3 Average access time

Even though I have not used Access time as a performance metric, I believe that as the cache size increases, the AAT should decrease(provided all other parameters remain constant).

7 Experimental Results

7.1 For WorkLoad 1

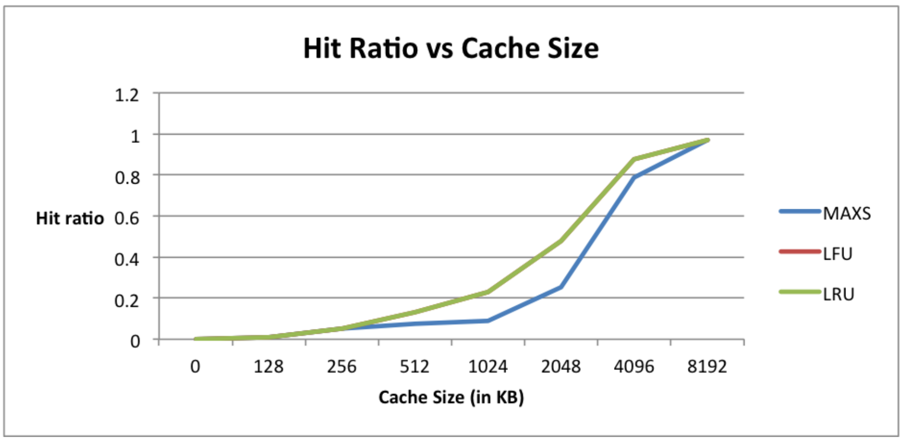

Fig. 1

As can be seen from the figure 1, LRU/LFU give better cache hit performance when compared to MAXS. Random also gives us a comparative performance (see Fig. 2). This is similar to what I had expected.

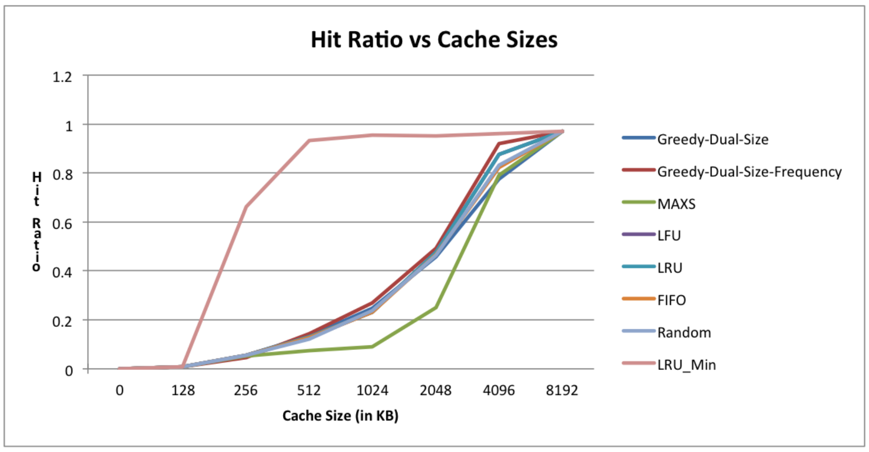

Fig. 2

Fig 2 shows the hit ratio performance of all the caching policies that I had implemented for varying cache sizes.

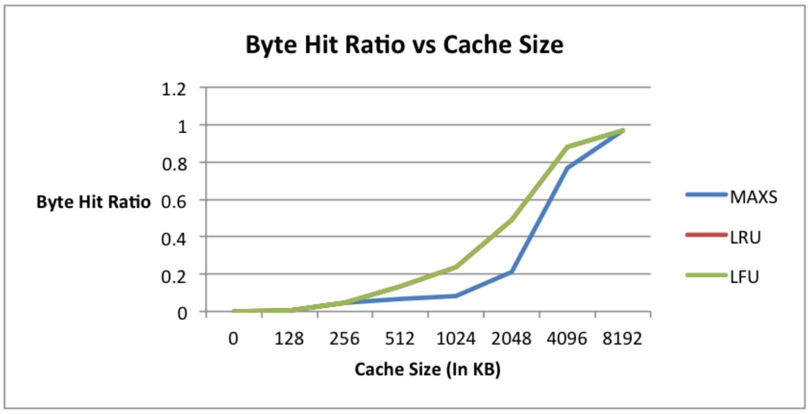

Fig. 3

As can be seen from the figure 3, LRU/LFU give better byte hit performance when compared to MAXS. Random also gives us a comparative performance (See Fig 4). This is similar to what I had expected.

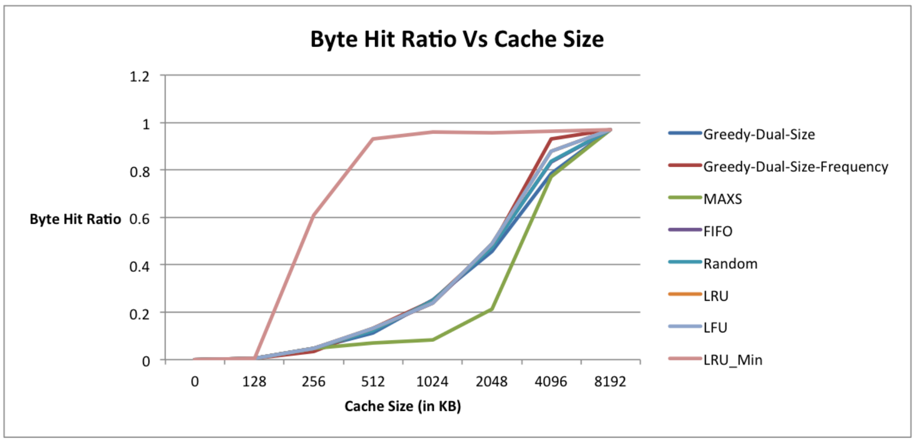

Fig. 4

Fig 4 shows the byte hit ratio performance of all the caching policies that I had implemented for varying cache sizes.

7.2 For WorkLoad 2

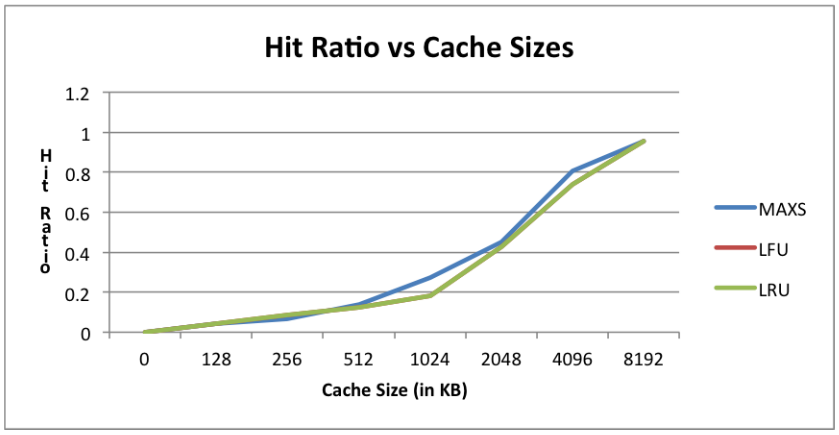

Fig. 5

As can be seen from the figure 5, LRU/LFU performs poorly in terms of cache hit performance when compared to MAXS. Random also gives us a comparative performance (See fig. 6). This is similar to what I had expected.

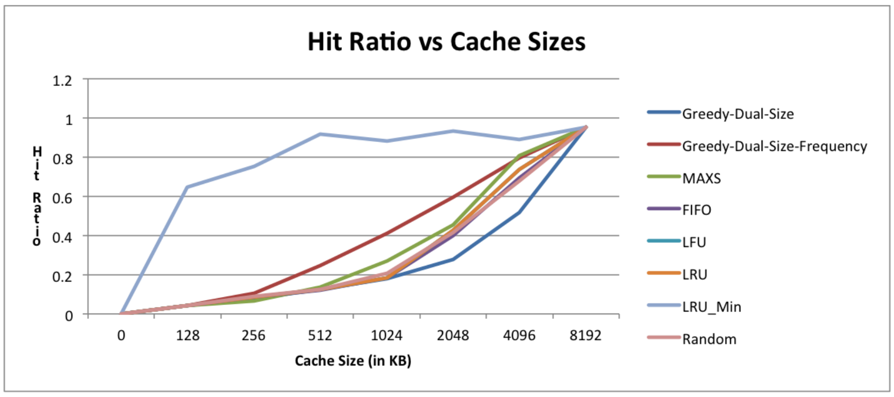

Fig. 6

Fig 6 shows the hit ratio performance of all the caching policies that I had implemented for varying cache sizes.

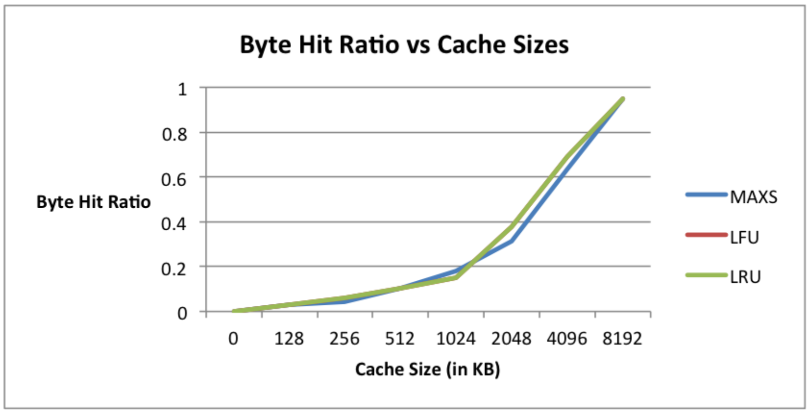

Fig. 7

As can be seen from the figure 7, MAXS and LRU/LFU give comparable performances and there is no better performer. Random also gives us a comparative performance(See fig. 8). This is similar to what I had expected.

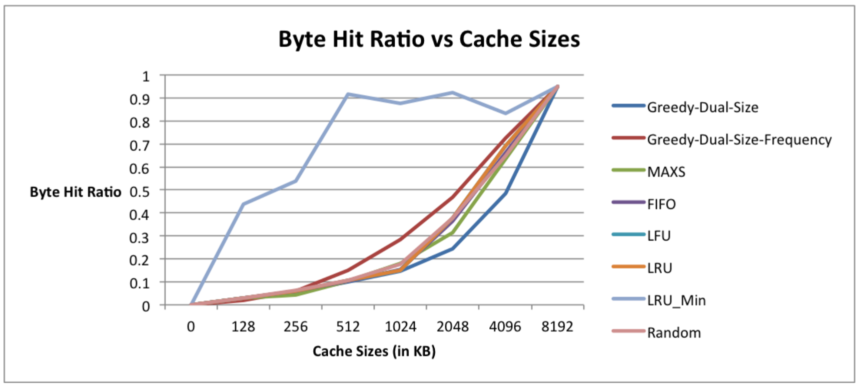

Fig. 8

Fig 8 shows the byte hit ratio performance of all the caching policies that I had implemented for varying cache sizes.

8 Analysis of Results

8.1 For WorkLoad 1

- workload 1 had a pattern where all the accesses consist of pages that are comparatively large in size and the difference in sizes of the pages used in the workload is not that huge.
- In such a scenario, Largest Size threw away those documents which are largest in size even though they are being accessed regularly. This resulted in repeated misses and thus reduce the hit ratio. A similar thing is observed for the byte hit ratio.
- In the same scenario, on the other hand, a LRU/LFU cache kept pages that are being accessed frequently , irrespective of their sizes. This lead to larger hit ratio and byte hit rate.
- Random also gave comparable results.

8.2 For WorkLoad 2

- Workload 2 is composed of groups of documents where there are 8 lightweight documents followed by 2 large size documents. The pattern consists of 100 such groups.

- In this scenario, MAXS removed the document with the largest size when a new document comes in whereas LRU/LFU remove documents based on lowest timestamp or frequency.
- Thus it can be seen from the graphs that MAXs gave better performance for workload 2 when compared to LRU/LFU.
- The byte hit ratio did not follow any exact pattern as expected and both types of caches performed well.
- Results of random , once again cannot be predicted.

9 Conclusion

- I was successful in creating RPC based proxy server using apache thrift.
- I also investigated and implemented different caching schemes for the server and evaluated the performance of the server under different load conditions with different caching schemes.
- I successfully created two completely different workloads on the basis of which I generated a hypothesis.
- I also proved our hypothesis true by carrying out several experiments.