Project Report
RPC-Based Proxy Server

Naman Goyal
Shantanu Deshpande

Introduction:

The objective of the project is to build a RPC-based proxy server. Remote procedure call is the communication between client and server connected through network. The functionality of RPC server is to get a URL from the client through a RPC call and return the text response back to the client. The major goal of the project is to implement various cache strategies on the RPC server for caching the URL response and experiment with them.

Caching is technique used in various system designs to store the frequently accessed data in location near to processing unit. For this project, we implement an in memory caching strategy that keeps the text response from various URLs stored in an in-memory structure. The cache is nearer to server than accessing the actual URL through the world wide web. So returning the responses using the cache should be order of magnitude faster. Any kind of cache is limited in size, so there should be some strategy to replace the items from cache in case of cache overflow. There are various different strategies, each of which perform differently in different scenarios. We implement three such strategies, **Random replacement**, **Least Recently Used** and **Least Frequently Used**. Details of these strategies is provided in following section of this report.

The final goal is to experiment these cache replacement strategies under different loads and give some analysis of performance.

We use **Apache Thrift framework** for simplifying the RPC implementation. It provides easy mechanism to define the service provided by the server and masks all the transportation details between server and client. We also use another library **libcurl,** which provides function to call HTTP get method.

Cache Design:

We need a generic cache design for various storing the webpages and then the cache replacement policies can be implemented over this data structure. We thought over various different structures for implementing cache data structure and finalized on **doubly linked list** backed by a **hash map** pointing to nodes through URL.

The structure for a node in a doubly linked list contains two pointers next and prev pointing to next and previous nodes in the list. We used doubly linked list because it's generic and provides O(1) insertion and deletion in the cache.

While implementing LRU, we get accessed entries to the front of the queue and the least accessed entries remain at the back of the cache structure.

A normal doubly linked list would provide O(N) lookup time, to make it faster we use **unordered_map** from C++ STL, which provides a constant lookup time and gives back the node by looking up the URL.

We also keep other state information like the **number of entries, cache size** in bits and the **vector** of all the URLs in the cache saved separately. The first two information is useful to check if the cache has exceeded the maximum size. The vector is used to get a random URL from the URLs saved in the cache. This is required since C++ map doesn't provide a random iterator and the vector provides the random iterator.

The operations supported by the cache structure are two basic operations **getPage()** and **putPage().** Both of these operation's implementation is provided by the particular cache replacement strategy. **getPage()** gets the webpage given the URL and returns true if there's an entry for the given URL otherwise false. **putPage()** puts the webpage in the cache. The implementation uses the hash map and vector to provide the implementation of these methods. The implementation should also take care of managing the memory associated with these nodes.

Cache Replacement Policies:

These policies are the heart of the project. We implement three following cache replacement strategies:

1) **Least Recently Used (LRU):**

The least recently used (LRU) cache policy replaces the page that is least recently used in the cache structure. In our implementation, we take out the accessed node from the doubly linked list structure and add it near the head. Thus the nodes near the head represent the most recently used pages and the nodes near the tail represents the pages least recently used. Thus when we reach the cache limit we can just throw out the page that is at the tail of the ache structure. This gives us O(1) get and put operations in LRU.

- Good Point: LRU works good if the load in the system has some locality, as in the page accessed has higher probability of accessing in near future, which might be the case in proxy server since people are more like to visit certain web

pages quite frequently. LRU would keep those pages mostly in memory and would decrease the access time.

- Bad Point: The performance of LRU depends on the locality assumption of the load. If the load doesn't follow this assumption there might be a negligible gain by keeping the recently used page in memory. But the operation itself might be costly since it would update the doubly list structure for every get and put operation.

  The other case where LRU might perform badly is if the cache size is smaller and not able to fit the pattern of the accesses. Say a user is accessing webpages in order A-B-C-D-A-B-C-D-A… but cache can hold just three pages, in that case LRU cache would keep evicting the required page and will result in really poor performance.

2) Random replacement policy: As the name suggest the policy is to just evict any of the page randomly from the cache structure.

The implementation of this requires to access any one of the URL in random order. This is not possible just by keeping the hash map structure since it doesn't provide the random iterator. So we keep a vector of all the URLs accessed till now and randomly pick one form it to delete. We use C++ random number generator to delete one of the nodes. The implementation is efficient and is O(1) for getPage() and putPage().

- Good Points: The get and put operations are simple and would require smaller time compared to LRU. It also doesn't assume anything about the pattern in the load, so it won't result in extreme bad performances. It might be a useful policy if there is nothing known about the workload on the system.

- Bad Points: For the normal workload like proxy server, some webpages are frequently accessed and this strategy might result in higher cache misses and would give poorer performance results on such loads.

3) Least Frequently Used policy: The Least Frequently Used (LFU) policy replaces the page which has been used the least frequently in the past. The assumption behind this is that a page which has not been used often in past will statistically be used less in future. The implementation of LFU uses an unordered hashmap which uses the web page url as key and the value is a data structure storing the web page HTML content and the frequency of usage of that webpage. So when we reach the limit of cache we search for the page with least frequency and remove it. The implementation

of this is O(1) for getPage() and O(1) for putPage() in best case and O(n) in worst case when the cache size is full.

- Good Point: The implementation is relatively simpler than LRU cache and similarly this policy would give good performance when there is fixed pattern of accessing some web page, which means that some page are accessed with higher frequency for certain time period. LFU would store such pages in cache and decrease the access time for the next access.
- Bad Point: The worst case performance of O(n) of putPage() may be one of the bad point of this policy if the new page being added to cache causes the cache limit to overflow and if this case happens often.

Evaluation Metrics:

We chose following two evaluation metrics for comparing the above mentioned cache strategies:

1. **Hit Ratio:** Hit ratio is a standard metrics to evaluate cache performance. The formula for hit ratio is the ratio of total pages found in the cache and total queries.

   Hit ratio directly evaluates the performance of cache since the ideal cache should require to go to the point of truth as less number of times as possible. Thus the higher the hit ratio, the better performing the cache is.

   Although the problem with the hit ratio might be that cache operations itself might be keeping a lot state and might be too costly, in that case only hit ratio is not enough to compare the cache performance.

2. **Response Time:** Since we are designing the service for the client and response time should be minimal in interactive application. We measure the **average response time** (in microsecond) of all the work load. The average response time is total response time for all the work load divided by the number of total queries.

   The lesser the average response time is, the better is cache performing. We will compare the average response time against all the cache policies. The response time might depend on various other situations like, the server load or the network contention. So we will try to perform our experiments multiple times under the same load and show the comparative results.

Workload Description:

Two different workload scenarios were chosen to compare the performance of the three above mentioned cache policies:

1. **Random Workload:** The random workload consisted of 25 different web pages which were each repeated a random amount of times to give a total of 150 web page request. A array of 25 different web pages was kept and each web page

was randomly chosen with replacement from the array for a total of 150 times to generate the random workload. This workload resembles the real-world workload where there is no fixed pattern in visiting the web pages.

2. **Repeated Workload:** The repeated workload consist of 25 different web pages which were each repeated 5 times to give a total of 150 webpages. From the 25 web pages, set of 5 random web pages were chosen 5 different times and then each web page in the set were repeated 6 in the random order. This represents the scenario where a user may be working on a particular task and may be switching between some web pages for the duration they are working on the same task. Then they may get another set of web pages when they start working on different task.

Experiment Description:

We set up the client and server on two different systems one on GT Wifi and oher on GT ethernet network. Both the systems are running Ubuntu 14.04 with intel i-7 processor. The server machine has 8GB of total RAM, so the entire cache fits in the virtual memory and should not be swapped during the experiments. The load (CPU, memory and network) on server machine is kept minimal during the experiment process.

The client program takes an URL filename as input and calls all the URLs given in the file. There are two such workloads generated as described in the previous section, one is random and other is repeated. Both of the workload, contains 150 URLs generated by some permutation (details in previous section) of given 25 URLS.

Thus there are two workloads and total 4 policies (no cache, LRU, Random, LFU) that we experiment with. We measure both the evaluation metrics mentioned for these 8 combinations. Also since response time is dependent on system and network load, we take the average of five such runs for all the 8 combinations.

We try varying the cache size for three values (64KB, 192KB and 256KB) for all the combinations mentioned above. Each of the size is smaller than total combined size of all the responses for each workload. This is to make sure that there are at least some evictions from the cache.

We expect the LRU and LFU to perform at least as good as random policy on the random workload. Since the implementation of both the policies is kept very minimal and both the getPage() and putPage() are constant time. We expect LRU and LFU to outperform random policy for the repeated loads since we have generated the repeated workloads such that there is some locality in the URL access.

Experiment Result:

The experimental results are provided in the graphs below. The graph in Fig.1 and Fig.2 show the variation in performance of hit ratio as the cache size is varied for three different policies for random and repeated workload respectively. Similarly Fig.3 and Fig.4 show the variation in performance of average response time as cache size is varied for the three different policies for the random and repeated workload respectively again.

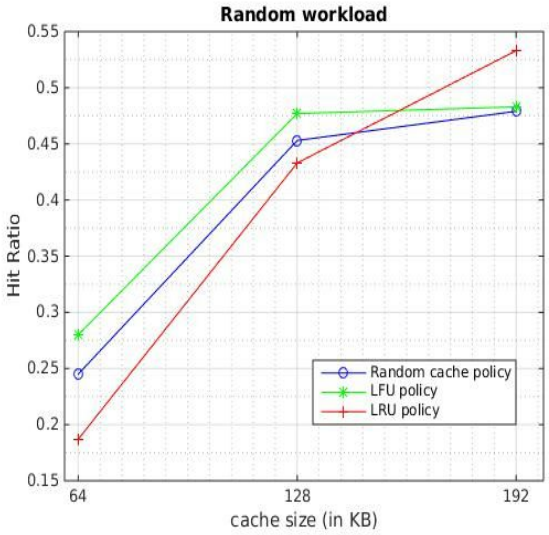

Fig.1 Cache Size vs Hit Ratio (Random)

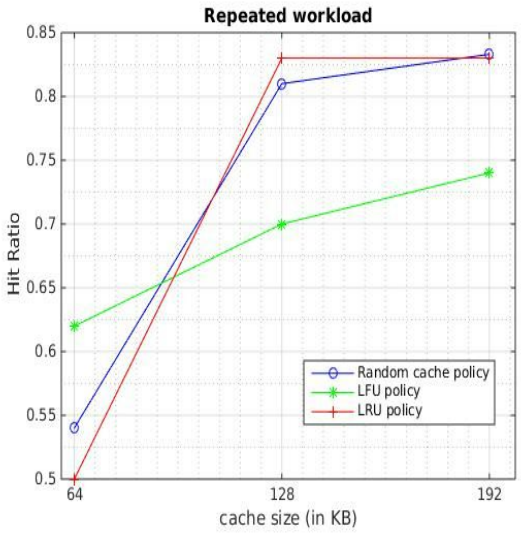

Fig.2 Cache Size vs Hit Ratio (Repeated)

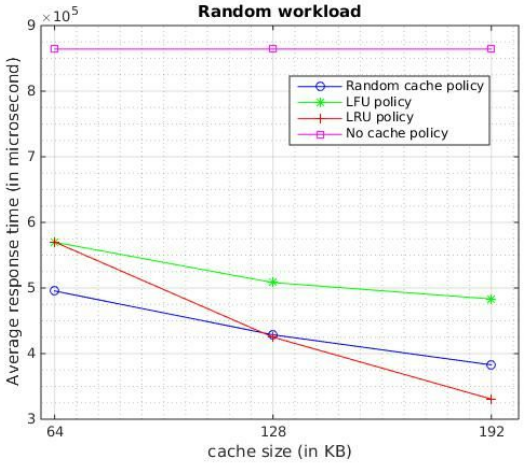

Fig.3 Cache Size vs Response Time (Random)

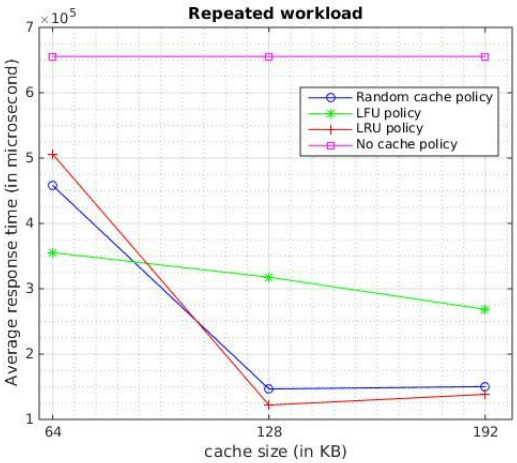

Fig.4 Cache Size vs Response Time (Repeated)

Result Analysis:

The four graphs presented above show the results of the conducted experiments. The first two graphs show the performance of variation of cache size on the hit ratio on two different workloads. The cache size is varied between 64 kb, 128 kb and 192 kb and the two different workloads are random workload and repeated workload.

We can observe in Fig.1 the variation of hit ratio versus cache size. As expected the hit ratio improves for all the three policies as cache size increases. This is expected as when the cache size increases, more web pages can be stored in the cache and hence the new request for the web pages can be serviced directly from cache which gives improvement in hit ratio. Between the three cache replacement algorithms, we observe that when cache size is smaller, random and LFU policy perform better than LRU policy. This can be attributed to the fact that when cache size is smaller, the largest web page consuming more memory may have been used least recently irrespective of their high frequency of usage and then those pages may be evicted to bring in new page. When the next time these page are visited they evict a lot of other smaller pages from the cache and this causes the hit ratio to reduce. But as the cache size increases, we observe that the performance of LRU policy improves and with sufficiently large cache LRU gives best performance.

In case of Fig.2 the variation of hit ratio versus cache size is shown for repeated workload for all three cache replacement policies. One of the first things to observe in this graph is that when the cache size is 192 kb, LRU and random policies give the optimum performance and they retain the web pages in cache so that for the request in repeated patterns, the web pages are already stored in the cache and hence the retrieval of the web pages happen from the cache instead of getting the request using curl. Another important feature in this graph is that the performance of LFU is comparatively worse than the other two policies even in case of maximum cache size. LFU does not show substantial improvement in hit ratio with increase in cache size for repeated load. This behaviour differs from our expectation of improved performance as cache size increases.

Fig.3 shows the performance of average response time for getting all the 150 webpages for both random workload as the cache size varies. We also ran the experiment with using any cache and as expected, the response time is worst in case of no cache as compared with any of cache replacement policies. Another point of observation is that as the cache size increases, the response time decreases irrespective of the cache replacement policy. This matches with our expectation as when the cache size increases, then more web pages are stored in the cache and hence when they are requested the next time, they can be directly accessed from the cache instead of using curl to get them from source. We can also observe that there is not a substantial decrease in response time in case of random workload the as the web page requested may be random and not be present in cache.

We can see in Fig.4 that the variation in response time as the cache size increases for the case of repeated workload. Again, we can observe that LRU and random policy give very good performance as the cache size increases. This is expected as with more cache memory, both the policies store all the required web

pages in the cache and hence the response time to get them the next time they are requested is negligible. This causes the average response time to substantial decrease with appropriate cache size. Also in case of LFU cache the response time is substantially smaller than other policies with smaller cache size but as the cache size increases, the reduction in response time is not much. This matches with the earlier graph of hit ratio versus cache size, where with increase in cache size the increase in hit ratio was not that siginificant as compared to other two policies.

Conclusion:

We have build a RPC based proxy server using thrift and curl and tested the performance of three different cache replacement policies, LRU, LFU, Random and no caching on two different workload scenarios, the random workload and the repeated workload. We present the performance in all cases of variance of either the cache replacement policies or the workload. From our experiments we could observe that -

1. When the appropriate cache size, the performance of the cache replacements algorithms are in the following order of best performance - LRU, Random, LFU and no caching. The performance of LFU can be improved with better implementation in which putPage() works in O(1).
2. In case of random workload, the performance of all cache replacement algorithms depends on the workload and the cache size. Also LRU gives better performance as the cache size increases.
3. With higher cache memory the response time and hit ratio can both be improved in case of any algorithms.