RPC-Based Proxy Server
An empirical study of Cache Replacement Policies

Manish Choudhary
mchoudhary8@mail.gatech.edu

Vaibhav Malkani
vmalkani3@mail.gatech.edu

March 26, 2014
Georgia Institute of Technology

1. Introduction

Remote Procedure Call (RPC) is a powerful and commonly-used abstraction for constructing distributed applications. Remote procedure calls are really useful in client-server architecture where a client can invoke the procedures published by the remote server easily just like the normal local procedure calls. Different technologies are available to build systems facilitating remote procedure calls. One of the modern RPC systems is Apache Thrift that provides efficient, extensible and cross language way to make remote procedure calls. The concept of remote procedure calls can be used for implementing proxies. A proxy server is basically a server acting as an intermediary for requests from clients seeking resources from the other servers. The web proxy servers can be used to access World Wide Web and also implement caching to provide faster access to the web pages. The web proxy servers are generally in a closer locality to clients and thus, avoid the network delays in going to the actual servers which may be located at far locations.

In this project, we have built a Web Proxy Server supported by remote procedure calls. Apache Thrift technology has been used to provide RPC mechanisms. The client has been provided with a single function that can be invoked to request web pages. The proxy server implements the mechanism to fetch the web pages using lib curl and returns the body of the document corresponding to the URL. This proxy server can be used to provide anonymity. In addition, it also facilitates the caching of the web pages so that the new requests for these pages can be served locally without going to the actual servers and therefore, avoiding the network and access delays involved.

The cache has been implemented as an in-memory cache and the cache size can be configured by passing command line arguments depending upon the memory available as per the server configuration. As the cache size can be limited, cache replacement policies have been implemented to evict the pages from the cache to create space for the pages fetched from the Web in response to the new requests. Three cache policies have been implemented to perform empirical study of these different policies based on the four different performance metrics under four different types of workloads.

As the name of the policies suggests, the “Radom Replacement” policy (RR) evicts pages from cache randomly; the “Least Recently Used” policy (LRU) evicts the page requested least recently and the “Largest Size First” (MAXS) removes the largest page first. To implement the mechanisms an abstract base class has been provided with common functions which is inherited and overridden by the separate classes for each policy type. The four metrics considered for performance evaluation of the cache policy are “Cache Hit Rate”, “Byte Hit Rate”, “Entry Removal Rate” and “Average Response Time”. “Cache Hit Rate” measures the rate or the fraction of the requested which are getting served from the cache and reflects how warm the cache are for a particular type of workload. “Byte Hit Rate”

tells the fraction of the total bytes returned in response to the client request from the cache and helps us to measure how much network traffic we are able to avoid because of the cache policy implemented. “Entry Removal Rate” reflects the number of pages which are getting evicted from the cache to make space for a new web page. This is important as it reflects the numbers of times the procedure for eviction is called and also how many different pages can be kept in the cache. Finally, “Average Response Time” measures the time in which client gets response for a requested URL averaged over a number of requests and helps in showing that implementation of caching really saves the time.

The four types of workloads have been implemented by reading the URLs to be requested from the text files using bash script. The workloads have been designed focusing on the characteristics of the different policies so that variable performance of the cache policies can be obtained and compared. The “No Repetition” workload has 30 unique URLs and gives a good picture of the “Average Response Time” of cache vs. the scenario without any cache implemented. It also helps in comparing the “Entry Removal Rate” of different cache policies. The “Anti-LRU” workload first fills up the cache to create good contention and then repeats a sequence of URLs. The total 90 URLs are requested create an environment in which LRU policy won’t be effective as the pattern requests pages somewhat in an order of least recent usage. The “Anti-MAXS” workload also fills up the cache to create a good contention environment and then alternatively requests the two largest pages already entered in the cache once. This shows that the “MAXS” policy may not be good for a workload in which only some large sized pages are accessed again and again as this policy removes the largest page from the cache first and thus, such a workload would result in very low cache hit. Finally, “Most Common” workload is similar to the daily based web access pattern of normal users i.e., some sites are accessed more frequently than the others in an intermixed fashion. It helps in analysing the cache policies in almost real scenarios.

The experiment has been conducted using two different machines as client and server both running Ubuntu Linux distribution as the operating system. The client machine has 4 different URL lists and 4 corresponding bash scripts to generate the four different workloads by requesting URLs from these different lists. The server has a published function “fetchWebPage” which is invoked by the client to request each URL. On the server, the main executable is “proxy_server” that takes cache size and policy type as input from the command line. The common functions for all the policies are implemented as part of an abstract class that is inherited by the specific child classes for each policy. The policy type argument helps in determining for which child class the object has to be created and to be assigned to an object pointer of the base class. We have maintained some class variables to keep the updated values for each performance metric which is finally printed at the end of each workload so that an aggregated result can be obtained. The “Average Response Time” is calculated on the client side by measuring the total time taken for each workload and then dividing by the total number of URLs requested.

The results are quite intuitive and match the speculations made by us while designing the workload. For the “No Repetition” workload, “No Cache” policy takes lesser time than cache policies implemented as other policies involve caching and replacing the pages which just becomes an overhead only. The “MAXS” policy results in least “Entry Removal Rate” which is intuitive as it removes the largest page first and thus, needs to evict less number of pages to create space for a new page. For the “Anti-LRU” workload, the “LRU” policy performs poorly and results in least “Cache Hit Rate” as it evicts the least recently used webpage and thus, the repetitive pattern forces it to fetch the web page from the Web only. The “Anti-MAXS” workload makes the “MAXS” policy perform poorly resulting in least “Cache Hit Rate” and “Byte Hit Rate” as the two URLs requested

alternatively are the largest pages and thus, one gets evicted whenever the other URL is requested forcing the proxy server to fetch the page from web next time it is requested. The “Most Common” workload doesn’t show objectively that one policy is better than the other. It depends upon the size of the pages requested and the particular pattern of the pages requested by the user. If the accessed pages don’t have much variation in the page sizes or the frequently accessed pages are among the largest ones, “LRU” may perform better than the others.

2. Cache Policy Description

The main reasons to add caching in web proxies are to reduce latency, reduce network traffic, and reduce server load. A request from client, if present in the web proxy cache will be closer to the client as opposed to the original server, assisting the reduction of load times to display content, which in turn, makes the website appear to be more responsive, and keeps expectancy at a minimum.

Caches are limited in size and only a limited amount of data can be stored within a cache at a time. Whenever the cache size is full and a new page is accessed, we need to put that page into the cache for which we have to remove some other pages from the cache to create sufficient space for the new entry. For caches to work efficiently we must ensure that the most relevant data is present in the cache at all the times. To achieve this we need to implement efficient cache replacement policies so that we keep the most relevant data in the cache at all the times. The cache policies considered for this experiment have been described below.

2.1. Random Replacement Policy

Whenever the cache is full and a new page is accessed, an index is obtained from a random number generator and the corresponding entry is evicted from the cache to allow the caching of the new page. The good point about this policy is that it involves the least algorithmic complexity i.e. we just need to generate a random index and evict the page at that index. On the contrary, this algorithm would not necessarily give us the variation in evictions with varying workload patterns i.e. we cannot tune this algorithm to our performance requirements of the cache.

2.2. LRU Policy

In this policy a timestamp is associated with each of the entries of the cache. Whenever an entry is put in the cache or an already present entry is accessed, the timestamp associated with that page is updated. The cache entry with the least timestamp will be the least recently used entry. Whenever the cache is full and a new page is accessed, the page with the least timestamp is chosen for eviction. The good point about this algorithm is that it helps in maintaining temporal locality i.e., it is good for workloads in which same pages are more likely to be accessed within short intervals. The bad part is that we need to update the timestamp of the page each time it is accessed i.e., even if the page lies in the cache, we need to modify the metadata associated with that page which results in slight increase in the complexity of the implementation.

2.3. MAXS Policy

In this policy a size field is associated with each of the entries of the cache. Whenever an entry is put in the cache, the metadata about the size of that page is also maintained. Whenever the cache is full and a new page is accessed, the page with the largest size is replaced. The good point about this algorithm is that the cache evictions drop significantly since by evicting the page with maximum size

we are creating a lot of space in the cache for future accesses. Specifically this policy is most suited for small sized caches .Thus we save a lot on the time spent on replacements. The bad part is that this policy does not preserve temporal locality. In a scenario where a large page is accessed in short durations, it is highly probable that the page would not be found in any of the future accesses and has to be brought into the cache each time.

3. Cache Design Description

We have followed an Object Oriented approach for our cache design. The base class is an Abstract Base Class that provides the basic interface and functions for the cache. The cache policies have been implemented by inheriting form the base class and overriding the virtual functions according to the replacement policies.

Whenever the server starts up , we pass the type of Cache Replacement Policy from the command line and make a new object according to the argument provided. We are utilizing Polymorphism provided by C++.All the functions are called by the base class pointer, which in turn points to the Cache Replacement Policy object in accordance with the selection of replacement policy made from the command line.

In the following sub sections the various classes and their members have been discussed.

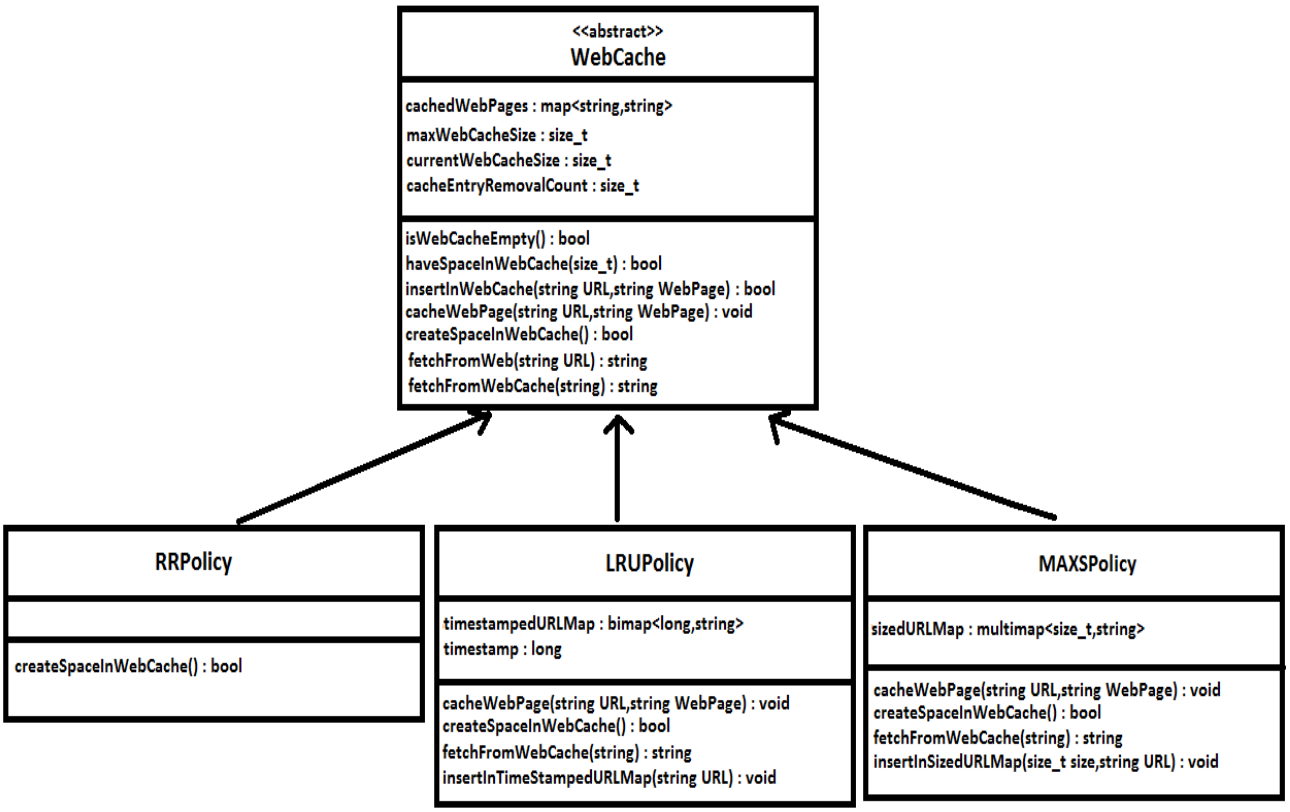

Figure 1: UML Diagram

3.1. WebCache Class

This is the base class in our cache implementation and we keep this class as an Abstract Base Class since we are not going to make any object of this class.

3.1.1. Member Variables:

- **cachedWebPages**

  This is the data structure that is used for storing the Web Pages. We are using an STL MAP for this purpose. The URL of the WebPage serves as the key and the body of the WebPage serves as the value. We could have made our implementation better by implementing an unordered MAP but since the C++ STL (except in C++11 which is not compatible with many compilers) provides only ordered maps we are using this implementation. Also using unordered maps may involve a larger memory print of the whole container.

- **maxwebCacheSize**

  This keeps the maximum size of the cache. We are taking a command line input for this parameter.

- **currentWebCacheSize**

  This keeps the updated current size of the cache at all the times.

- **cacheEntryRemovalCount**

  This maintains the cumulative count of the total number of entries removed from the cache during the whole run.

3.1.2. Member Functions:

- **isWebCacheEmpty()**

  Whenever a client requests a page, first this function is called to check if the cache is empty. If true, then the web page is directly fetched from the web server instead of looking up in the cache for that page. Using this function saves the time in calling the function for look up into the cache.

  The function checks if a cache is empty by using the empty() function of the MAP STL since we are keeping all our cache entries in a MAP STL. The complexity of this operation is O(1).

- **haveSpaceInWebCache()**

  This function checks whether the total size of the cache exceeds the total size of the cache and then accordingly calls the function for page eviction and page insertion. The current size of the cache is maintained in **currentWebCacheSize** member. The maximum size of the cache is maintained in **maxwebCacheSize** member**.**

  The function returns TRUE if

  currentWebCacheSize + webPageSize > maxwebCacheSize

  where **webPageSize** is the size of the page that has to be inserted into the cache.

- **insertInWebCache()**

This function inserts the Web Page into the cache and increments the current size of the cache. For insertion into the cache (which is a MAP STL),the URL is used as the key and the Web Page as the corresponding value. The complexity for this operation is O (log (n)).

- **fetchFromWeb()**

In this function the Web Page is fetched corresponding to a given URL. The Libcurl library has been used for performing this task. The implementation for this part has been adopted from the Project Specifications provided at T-Square with some modifications.

- **cacheWebPage()**

In this function we check if there is space left in the cache. If not we keep calling the function **createSpaceInWebCache()** until space becomes enough to keep that Web Page in the cache. After this we call the **insertInWebCache()** to insert Web Page into the cache.

- **fetchFromWebCache()**

In this function we are doing a lookup into the cache to see if the entry is already present. If present we return the Web Page otherwise an empty string. We are using the find () function of the MAP STL to do the look up. The complexity of this function is O (log (n)).

- **createSpaceInWebCache()**

This is the function which determines which entries to evict. We are just giving an interface for this function in the base class. The actual implementation is provided in the derived classes i.e. the classes implementing the cache replacement policies.

3.2. RRPolicy Class - Random Replacement Policy

3.2.1. Member Variables

All member variables of this class are inherited from the base class.

- **createSpaceInWebCache()**

This is the function responsible for eviction from the cache using Random Replacement Policy.In this function, we advance the iterator of the MAP STL by a random number which is generated by using the rand() function and erase that entry from the MAP. We are using the **advance()** and **erase()** function provided by the MAP STL which perform the task in O(log(n)) and O(1) respectively. **Erase()** takes O(1) in this case as the iterator is already pointing to the entry to be erased. So the complexity of Cache Eviction is O(log(n)) in this algorithm.

Note: This is the only function that is overridden by Random Replacement Policy derived class.

3.3. LRUPolicy Class - Least Recently Used Policy

In LRU we implemented two maps. The first one is to map URL to WebPages and the second one to map timestamps to URLs. For the latter we used Boost bi-maps the reason for which is explained below.

3.3.1. Member Variables

- **timestampedURLMap**

  This data structure is used to keep the timestamps associated with the all the URL stored in the cache. We are using Boost bimap for this. In this bimap timestamp and URL serve as both key and value. The reason for using bimap is that we need to use both the members of the Map as key and value. To evict the webpage we need to make timestamp the key so as to find the least timestamp. Also we need to use the URL as a key while updating the timestamps of the URL already present in the cache. We could have used two individual Map data structures for this purpose but went with bimap considering its more efficient implementation.

- **Timestamp**

  The timestamp does not store the current time but behaves as a counter. Each time a new Web page is fetched from the server or from the cache the value of this timestamp variable is incremented and stored in the timestamp corresponding to that URL in the bimap.

3.3.2. Member Functions

- **createSpaceInWebCache()**

  This is the function responsible for eviction from the cache using LRU Policy. In this we search for the URL at the beginning of the bimap(since bimap is also ordered according to the left key).The first element of the bimap corresponds to the minimum timestamp and hence the Least Recently Used page. After retrieving the URL we use this as a key to delete the WebPage from the cachedWebPages map(the map that stores the cached web pages).The complexity of eviction using this function is O(log(n)).

- **cacheWebPage()**

  This function overrides the implementation of the function in the base class. This is due to the fact that while caching a page in the cache we need to update two mappings. One from URL to WebPage and other from timestamp to URL.So this function calls the two functions insertInTimestampedURLMap() and insertInWebCache() respectively. The latter function's implementation is the same as that in the base class i.e. it is not overridden here. The complexity of inserting the WebPage in the cache and maintaining the associated meta-data(i.e. timestamp) is O(log(n)).

- **insertInTimestampedURLMap()**

  This function inserts the timestamp and URL pair in the bimap. The complexity of insertion is O(log(n)).

- **fetchFromWebCache()**

This function also overrides the implementation of the function in the base class. This looks up for the URL in the cache and also updates the timestamp associated with that URL in the bimap. The complexity of this operation is also O(log(n)).

3.4. MAXSPolicy Class - MAXS Cache Replacement Policy

3.4.1. Member Variables

- **sizedURLMap**

This data structure is used to maintain a mapping of the size of the page to the URL.We are using a MultiMap for this purpose. The reason for using a MultiMap is that the size of more than one Web Pages can be identical. So we cannot use Map for our purpose here.

3.4.2. Member Functions:

- **createSpaceInWebCache()**

This is the function responsible for eviction from the cache using MAXS Policy. The URL with the largest Web Page size is located at the last element of the MultiMap (since MultiMaps are sorted).After finding the URL ,we use it as a key to delete the Web Cache from the cachedWebPages data structure. The complexity of this operation is O(log(n)) since it takes O(1) to delete from the sizedURLMap data structure to find the URL and O(log(n)) to delete the Web Page corresponding to that URL in cachedWebPages.

- **cacheWebPage()**

This function overrides the implementation of the function in the base class. This is due to the fact that while caching a page in the cache we need to update two mappings. One from URL to WebPage and other from size of Web Page to URL.So this function calls the two functions insertInSizedURLMap() and insertInWebCache() respectively. The latter function's implementation is the same as that in the base class i.e. it is not overridden here. The complexity of inserting the WebPage in the cache and maintaining the associated meta-data(i.e. Web Page size) is O(log(n)).

- **insertInSizedURLMap()**

This function inserts the Web Page size and URL pair in the MultiMap. The complexity of insertion is O(log(N)).

4. Metrics for Performance Measurement

To perform the empirical study of the different policies under consideration following performance metrics have been selected.

4.1. Hit Rate

Hit Rate is the ratio of the number of times a Web Page request was found to be in the cache to the total number of Web Page requests. It is a metric of the proportion of documents that have been served from the cache instead of the original server and it gives an idea of the reduction of the latency observed by the users.

Increase in Hit Rate indicates that a given cache eviction policy is performing better for that particular workload since even after evictions from the cache more relevant pages are present in the cache with higher probability. The hit rate is a more reasonable measure of effectiveness if the Web Pages are homogenous in size.

4.2. Byte Hit Rate

It is defined as the ratio of the summation of the document sizes that cause a hit in the cache and the total size of all the web pages accessed. BHR is a metric of the proportion of traffic that has been served by the cache instead of the original server and it shows the bandwidth saved by the cache.

Byte Hit Rate gives us an idea of how much we are saving on the network bandwidth .Higher BHR means that we are getting more number of bytes from the cache and thus saving the network bandwidth between the Web proxy and the Web Server.

Also it is worth mentioning that since in this project we are only fetching the body of the Web Page and not its headers, the actual BHR should be higher than the observed BHR. Increase in BHR also indicates better performance of a Cache Replacement Policy for a particular workload.

4.3. Entry Removal Rate (Eviction Rate)

This is the ratio of the number of times an entry is evicted from the cache to the total number of accesses. Most algorithms work under the assumption that disk accesses or CPU time is far less than the network latency needed to send the data to the client. But the time needed to run the replacement algorithms may be comparable to the network latency to the client in quite a few cases. In these cases this metric becomes important.

Also higher removal rate indicates that it is highly probable that high frequency requests would not find the Web Pages in cache resulting in decrease in performance. Lower Entry Removal Rate indicates better performance of a Cache Replacement Policy for a particular workload.

4.4. Average Response Time

This is the total time taken for a given set of requests to complete divided by the number of requests in that set. This gives us an idea about the latency observed by the users. It should be however noted that this metric need not necessarily indicate the performance of a Cache Replacement Policy since the access times from the Web Server and even the Proxy Server show bursty patterns due to many reasons such as collisions in wireless medium, firewalls checking for malicious activity in the Web Servers and so on.

As further described in our experimental results below, a few URLs were showing very high variance in consecutive access times from the Web Server.

Note: Although these metrics are related, optimizing one may not necessarily optimize the other.

5. WorkLoad Design

We designed a total of four workloads to test the performance of different cache replacement policies.

5.1. "No Repetition" Workload

In this work load we have all the URLs as distinct.

A B C D E F G H I J K L M N O P……

We designed this workload to see the latency caused due to the replacement policies implemented in the cache. Since all the URLs are distinct it is evident that there will be no hits in the cache but there will be replacements for sure as we have made sure that the total size of all the Web Pages requested exceeds the maximum size of the cache. So this workload helps us distinguish the best cache replacement policy from the point of view of the total delay caused by the replacement policies.

For our experiments, we hard code a list of 30 distinct URLs after carefully choosing them so that the total request size exceeds the maximum cache size by at least 3-4 times.

5.2. "Anti-LRU" Workload

A B C D E F A B C D E F A B C D E F….

We designed this workload with the goal of observing a worse performance of LRU replacement policy in comparison to the other policies. We carefully selected some number of distinct URL so that their cumulative size just exceeds the maximum cache size. As we can see from the above pattern if the cache has no more space to insert F it will replace the least recently used URL i.e.A but then in the very next access A would not find itself in the cache and will evict B to get in. The same trend continues and a very poor performance is observed for LRU policy. Other cache policies will outperform LRU for this work load.

For our experiments, we hard code a list of 90 URLs after carefully choosing them. A group of 15 distinct URLs in chosen such that it just exceeds the cache size on the 15^{th} access. After that this list of 15 URLs is repeated to form 90 URLs.

5.3. "Anti-MAXS" Workload

A B C D E F A B A B A B….

We designed this workload with the goal of observing a worse performance of MAXS replacement policy in comparison to the other policies. We choose A and B to be URLs with large sized Web Pages and carefully select the URLs so that the cache size just exceed on the access of F. When F is accessed A will be replaced (since it is the biggest sized URL in the cache).But immediately after this A will be accessed and would lead into a miss and yet again an increase in the cache size beyond the maximum limit. This will cause the replacement of B and the trend would continue. Other cache policies will outperform MAXS for this work load.

For our experiments, we hard code a list of 90 URLs after carefully choosing them. A group of 15 distinct URLs in chosen such that it just exceeds the cache size on the 15^{th} access. After that the two URLs with the biggest Web pages are accessed alternatively as shown in the above pattern.

5.4. "Most Common" Workload

Let A and B be the most frequently accessed URLs. The pattern for this workload looks like

A B C A D B A B E F G A B B A…

We designed this workload with the goal of analyzing the performance of the various cache policies on a workload that is practically more common. For our experiments, we hard code a list of 90 URLs after carefully choosing them. We choose certain common URLs like www.google.com and www.youtube.com and place them frequently in between not so frequently accessed URLs.

Note: In all the workloads we have ensured that the total size of the Web Pages of all the accessed URLs exceed well above the maximum cache size.

6. Experiment Description

6.1. Experiment Bed

The experiment required client-server architecture to execute and analysis the system built. Both the client and the server were different machines with running Ubuntu Linux distribution as the operating system. Apache Thrift and other dependencies were installed on both the machines to enable RPC mechanism. Lib Curl was installed on server to allow the server to use this technology to fetch the requested pages from the web as per the need. Both client and server were kept close to each other and were connected via Georgia Tech's Wi-Fi.

6.2. Experiment Setup

To perform the experiment, four different workloads were designed with reasoning as explained in the above sections. We created a separate text file containing the list of URLs for each workload. Also, a separate bash script was developed to read each list of URLs from the text files and invoke the function "fetchWebPage()" published by the proxy server to be used by the clients to request the web pages by providing the URL as the argument.

On the server, the main executable is "proxy_server" that takes "Max Cache Size" and "Replacement Policy" as the inputs from the command line. The first input helps in making the cache size configurable. All the common functions to support the cache mechanism have been implemented in "webCache.cpp" which acts as the abstract class. Each policy has been implemented as a separate class which inherits the "WebCache" class and uses some of its functions while providing its own implementation for the other functions as per the need. The "Replacement Policy" command line argument helps in determining for which child class the object has to be created and to be assigned to an object pointer of the base class. The function "fetchWebPage()" is invoked on each request which in turn calls the different functions as explained in above sections as per the "Replacement Policy".

The performance metrics have been measured in the following way:

- **"Cache Hit Rate":** A variable named "cacheHits" of long type is maintained and initialized to zero. It is incremented by one every time a requested page is found in the cache. Finally, the total count is printed after completing each workload. This total count of the hits in the cache is divided by the total number of requests to get the cache hit rate.

- **"Byte Hit Rate":** To measure the byte hit rate two long variables "totalBytesReturned" and "bytesReturnedFromCache" are maintained and initialized to zero. The size of the web page to be returned in response to any client request is added to the variable "totalBytesReturned" and the size of the response web page is added to "bytesReturnedFromCache" in case the page to be returned is found in the cache. At the end of each workload, dividing "bytesReturnedFromCache" with "totalBytesReturned" gives the fraction of the total bytes which are returned from the cache.

- **"Entry Removal Rate":** To measure this performance metric a class variable "cacheEntryRemovalCount" is maintained for each policy. This variable is initialized to zero and is incremented every time an entry is removed from the cache i.e., every time "createSpaceInWebCache()" function is called. The final value of this count variable is printed at the end of each workload which is divided by the total number of requests to obtain an average number of entries to be evicted to create space for a new web page.

- **"Average Response Time":** The average response time is the average time in which the client gets the response back from the server for any requested URL. This metric has been measured on the client side as it makes more sense to look at it from a client's perspective. To measure this, the code has been added in the bash script to take the time at the start and the end of the URL list for each workload. It gives the total time taken by the server implementing a particular cache policy to respond to all the URLs requested. This time is divided with the total number of requests to obtain the average response time.

6.3. Experiment Execution

The experiment has been conducted for each type of workload by running the corresponding bash scripts. Each workload is tested for all the three policies implemented as well as for the "No Cache" policy by setting the cache size to zero. In addition, the cache sized is varied for each policy. The performance metrics have been measured for the cache size 410 KB, 1024 KB, 1524 KB and 2048 KB for each policy under each workload.

6.4. Hypothesis –The expected Results

The workloads have been designed focusing on the characteristics of different replacement policies. The four different types of workloads are expected to bring out the different performances and the behaviour of the policies and thus, lighting the good as well as the bad aspects of each policy under consideration for the empirical study. The expected results for each workload are as follow:

- "No Repetition" Workload
    - There should not be any cache hit for any active cache policy on the server i.e., the "Cache hit Rate" should be zero as all the URLs are unique.
    - The "Byte Hit Rate" should also be zero as no page should be found in the cache.

    - The “Entry Removal Rate” should be least for the “MAXS” policy as it evicts the largest pages first and thus, would have to remove less number of pages to create space for the new web pages to be cached.
    - The “Average Response Time” should be minimum in case of “No Cache” policy as there is no cache hit for any policy implemented. So, all the web pages have to be fetched from the web for any of the policies. All other policies except the “No Cache” policy would have to perform management operations such as cache look up, insertion of web pages and eviction of cached web pages which should take more time. Therefore, “No Cache” policy should have the least value for this metric.
- “Anti-LRU” Workload
    - The workload has been designed to generate a pattern that should result in an overall bad performance of “LRU” cache replacement policy.
    - “RR” policy and “MAXS” policy should perform better than the “LRU” policy.
    - The “Cache Hit Rate” and “Byte Hit Rate” should be least for “LRU” policy as the page requested would already have been evicted and thus, would have to be fetched from the web.
    - The “Entry Removal Rate” should be high for “LRU” as the workload has been designed in a way such that it would be forced to evict pages to insert new entries in the cache.
    - “MAXS” should have least “Entry Removal Rate” as it replaces the largest pages first and thus, would have to evict lesser number of pages to allow new pages to be cached.
    - Nothing can be said about the “RR” policy as the behaviour would depend upon the particular random behaviour at the execution time.
    - The “Average Response Time” of the “LRU” policy may be higher than the “No Cache” policy as the hit rate for “LRU” policy would be very less and would have to fetch most of the pages from web and the cache management operations would become an overhead adding much to the response time.
- “Anti-MAXS” Workload
    - The workload has been designed in a way such that “MAXS” policy should perform worse than the other policies.
    - The “Cache Hit Rate” and “Byte Hit Rate” should be less for the “MAXS” policy as it would evict the largest page and would have to fetch it from web the next time this page is requested.
    - The “Entry Removal Rate” should be higher for “MAXS” policy as the two largest pages are requested alternatively and thus, eviction would happen for each such request resulting in high eviction rate.
    - “RR” policy and “LRU” policy should be perform better than “MAXS” policy for this particular workload.
    - Because of high “Entry Removal Rate” and low “Cache Hit Rate” the “MAXS” policy should take more time than the “No cache” policy.
- “Most Common” Workload
    - This workload is completely experimental workload to observe the behaviour of different cache policies.
    - The workload is a mix of frequently accessed web pages, some other web pages and some large size web pages.

- Expected behaviour is better performance of “LRU” and “MAXS” policies as compared to “RR” policy.
- “MAXS” should have the least “Entry Removal Rate” because of the eviction of the largest pages and thus, availability of sufficient space after less number of evictions.
- Performance results for all the three policies should show the benefit of using cache. The average response time for these policies should be less than that of “No Cache” policy.

The above mentioned behaviour is expected if there is contention in the cache. If there is sufficient space in the cache to keep all the web pages, the contention would be less (or may be zero) and thus, all the policies should perform well in such scenarios.

7. Experimental Results and Analysis

The results obtained by executing the experiment have been presented below. The results have been categorized on the basis of the type of workload where each subsection covers the results for performance metric comparing the results for the different cache policies implemented.

7.1. “No Repetition” Workload

This workload is a set of unique URLs designed to focus on the “Average Response Time” and the “Entry Removal Rate” as it can give a clear picture of these parameters with respect to each policy. The workload consists of 30 unique URLs.

7.1.1. Cache Hit Rate

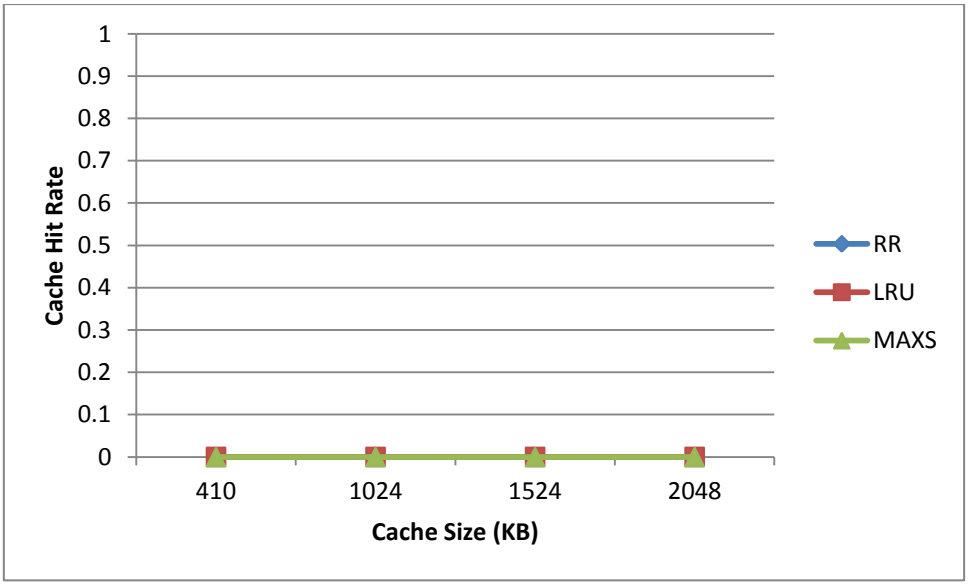

Figure 2: Cache Hit Rate vs. Cache Size

The Cache Hit Rate doesn’t apply for “No Cache” Policy as in that case the cache size is set to zero. The graph presents the results for all the other three policies implemented. The Cache Hit Rate always remains zero across all the cache sizes for all the policies.

Analysis

This result is quite intuitive as the URLs are unique there is no benefit of keeping web pages in the cache. Each request is new and thus, gets fetched from the Web. In other words, no request can be served from the cache as the requested page being a completely new request is never found in the cache. Hence, it can be concluded that the results matched our hypothesis. Increasing the cache size has no impact on this behaviour as it doesn't matter when we don't get a hit in the cache.

7.1.2. Byte Hit Rate

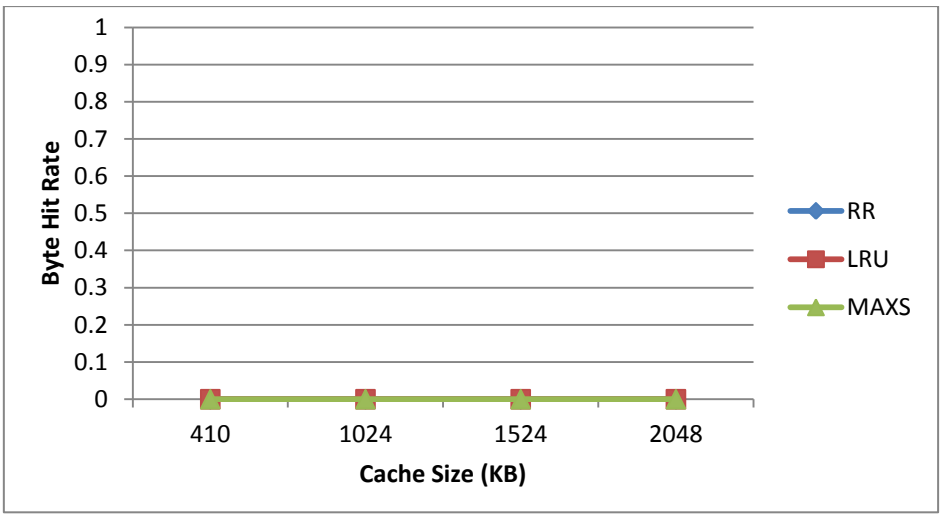

Figure 3: Byte Hit Rate vs. Cache Size

The Byte Hit Rate measured for all the policies expect "No Cache" policy as it doesn't apply to the latter. The results show that the Byte Hit Rate is zero for all the three policies across all the cache sizes.

Analysis

This is also intuitive as the Cache Hit Rate is zero. The number of bytes returned from the cache would always be zero. All the requests are served from the web and therefore are the bytes have to be transmitted over network and we are not able to get the benefit of caches to avoid transmission of the bytes across the network.

7.1.3. Entry Removal Rate

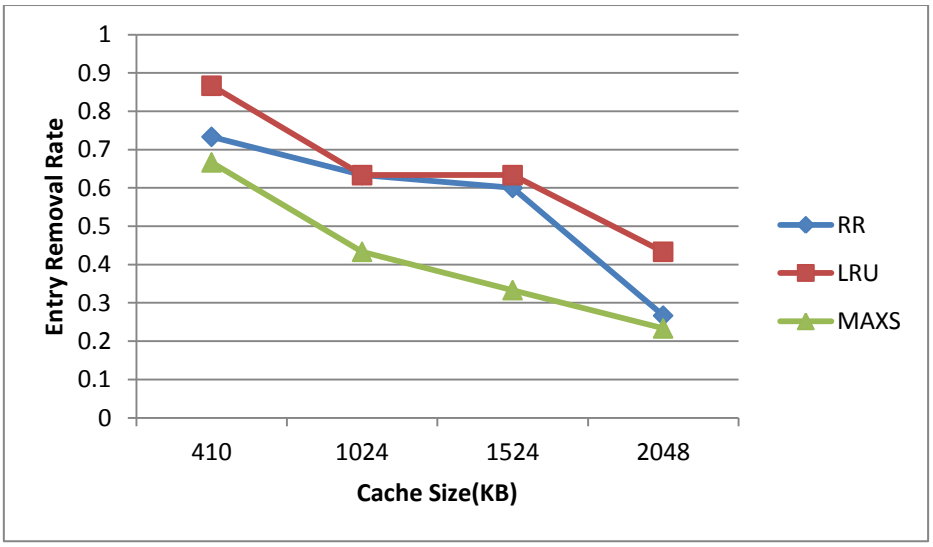

Figure 4: Entry Removal Rate vs. Cache Size

As explained in above sections, the Entry removal Rate reflects the average number of pages which get evicted from the cache to create space for the new web page to be cached. The results show that the "MAXS" policy has the least Entry Removal Rate among all the policies implemented. In addition, the Entry Removal Rate drops for all the policies as we increase the cache size. Again, "No Cache" Policy is not displayed in the graph as the performance metric doesn't apply in this case.

Analysis

The results match our hypothesis and can be derived theoretically also by studying the characteristics of each policy implemented. The "MAXS" policy results in the least Entry Removal Rate as we evict the page with the largest size and thus, would have to remove less number of pages to create space for a new web page. The "RR" policy choses some random page for eviction and thus, may require to remove more entries from the cache. Similarly, the eviction candidate in case of "LRU" policy is the least recently used and thus, cache may not have sufficient space to keep the new web cache even after removing one entry. So, the key point is that the cache replacement policy that considers the size of the pages for eviction would likely have a smaller Entry Removal Rate as the calls to the procedure responsible for removal of an entry would be less.

"RR" policy shows a better performance than the "LRU" policy which was not predicted before. The reason for this may be the random behaviour of the "RR" policy. As it chooses random pages for eviction, it is possible that sometimes it choose some page which is not least recently used and has a larger size. So, on an average, it would showcase a better performance than "LRU" policy which always chooses the least recently used page irrespective of the size of the page and thus, would have to evict more pages in case the less recently used pages have smaller size. Therefore, the size of the web pages is really the governing factor in performance of the replacement policies for this particular metric.

In addition, as we increase the size of the cache, the contention decreases and we can keep more pages in the cache without evicting pages from the cache. Therefore, the performance increases across all the policies for this performance metric.

7.1.4. Average Response Time

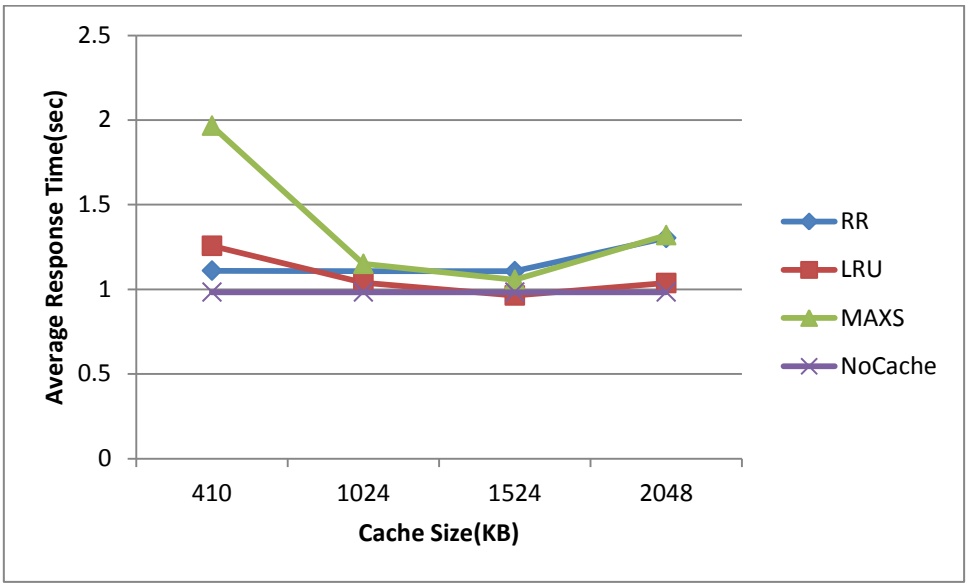

Figure 5: Average Response Time vs. Cache Size

The average time has been measured from the client side. The results show that the "No Cache" policy takes the least average time to respond to the client requests. The results for all other policies vary.

Analysis

The results match our hypothesis as all the cache replacement policies perform worse than the "No Cache" policy. As the results show, the "No Cache" policy has the least average response time for all the cache sizes. This is also quite intuitive as we are not able to exploit the caches because the Cache Hit Rate is zero. So, for each request, all the three implemented policies have to fetch the page from the Web just like the "No Cache" policy. In addition to this, the three policies have to spend much time in management operations such as cache lookup, insertion of new entries in the caches and eviction of pages from the cache to create space for the new pages fetched freshly from the web.

This management of cache just become an overhead and adds to the latency in the response. Therefore, it can be concluded that in such a scenario, maintaining caches is not fruitful and "No Cache" policy would be a better option.

Note: This workload shows that the implementation of caches is not a good idea. But, this workload is not very practical (imagine a user accessing a new URL every time). This workload has been considered only for checking the integrity of the implementation as well as the displaying the key characteristics of the policies with respect to response time and the Entry Removal Rate.

7.2. “Anti-LRU” Workload

This workload has been designed to show a scenario in which case the “LRU” replacement policy performs badly. The total number of URLs requested as part of this workload is 90. The workload first warms the cache by requesting some pages and then requests a repeated set of URLs.

Note: “No Cache” policy is shown in the graph only for Average Response Time metric as other metrics are not applicable in this case.

7.2.1. Cache Hit Rate

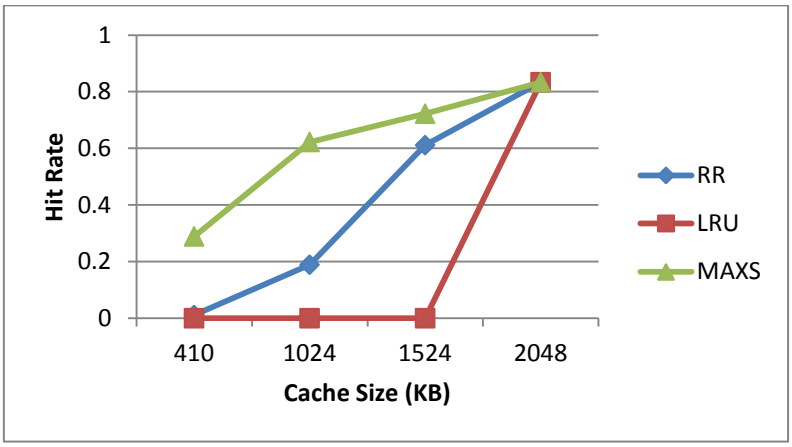

Figure 6: Cache Hit Rate vs. Cache Size

The results display that the “LRU” policy performs worst until the cache size reaches to 2048 KB. The “MAXS” shows the best performance among all the three policies implemented. Finally, all the policies perform well when the cache size is set to 2048 KB.

Analysis

Some of the results match our predictions. The “LRU” cache replacement policy doesn’t work well in this workload. The “LRU” policy replaces the least recently used page first. Our workload requests the pages in an order in repeated manner. By the time a page is requested again, it has already been replaced by the policy to create space for the other requested pages. So, it leads to a cache miss and proxy server is forced to fetch the page from the web. The “MAXS” policy performs well as it focuses on the page size rather than the usage time and thus, finds the cache warm for some of the repeated requested.

The comparison between “RR” policy and “MAXS” policy was not predicted before. It is possible because of the random nature of the “RR” policy leading it to evict the less recently used pages sometimes and thus, resulting in a performance worse than the MAXS policy. Another reason could be the fact that “MAXS” removes the largest sized page and thus, can keep more number of entries in the cache resulting in higher cache hit rate.

In addition, when the cache size is less, there is a lot of contention and only some pages can be kept in the cache leading to smaller hit rate. As the cache size increases, more pages can be kept at one time resulting in the increase of hit rate. One point to note is that the “LRU” policy performs equivalent to other policies for 2048 KB cache. It is because in this scenario, the total size of all the

responses for the repeated requests becomes comparable (which was much more before) to the cache size and thus, most of the requests can be found in the cache.

7.2.2. Byte Hit Rate

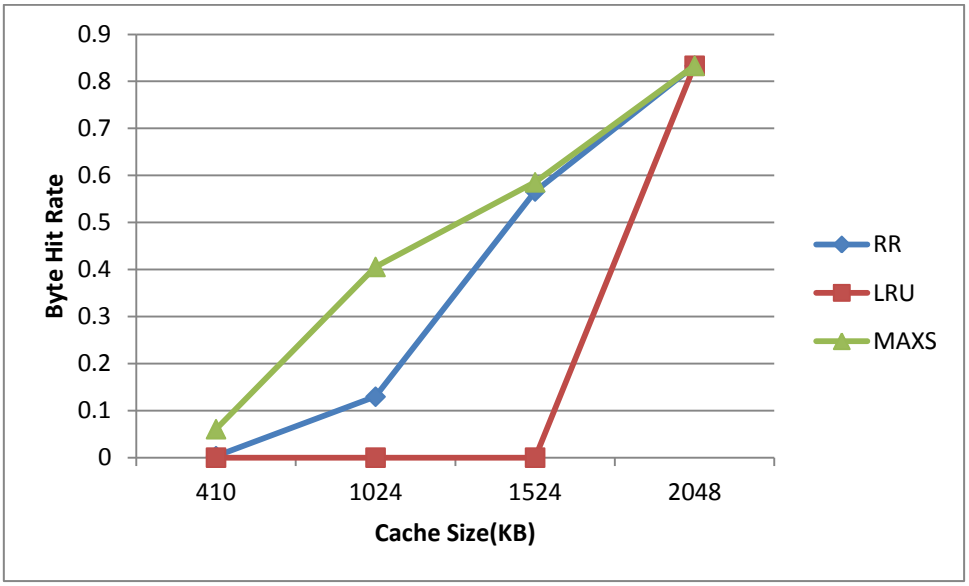

Figure 7: Byte Hit Rate vs. Cache Size

The "MAXS" policy performs best for all the cache sizes. The "LRU" performs worst with returning zero bytes from the cache until the cache size reaches 2048 KB. The results indicate that the performance of the "RR" policy is definitely better than the "LRU" but performs poorly as compared to "MAXS" especially when there is much contention in the cache.

Analysis

This is an expected behaviour for "LRU" policy as it doesn't get any hit for almost all the requests and thus, no bytes are served from the cache until it has sufficient space to keep almost all the requests. The "MAXS" policy has a high hit rate and thus, can serve many requests from the cache.

In case of 410 KB sized cache, MAXS has byte hit rate somewhere around 0.05 even though the cache hit rate is around 0.3. It may be because of the fact "MAXS" is getting hits for small sized pages as it removes the largest sized page. So, even though the hit rate is good, the bytes returned from the cache are limited as it serves only the smaller pages from the cache and has to fetch the large pages from the Web.

As the cache size increases, both "MAXS" and "RR" policies perform better as they can keep more entries in the cache. "LRU" policy can also keep more entries as the cache size increases but because of the nature of the workload and the eviction of the least recently used web page, it won't perform well until it has sufficient space to keep almost all the webpages corresponding to the repeated URLs in the list.

7.2.3. Entry Removal Rate

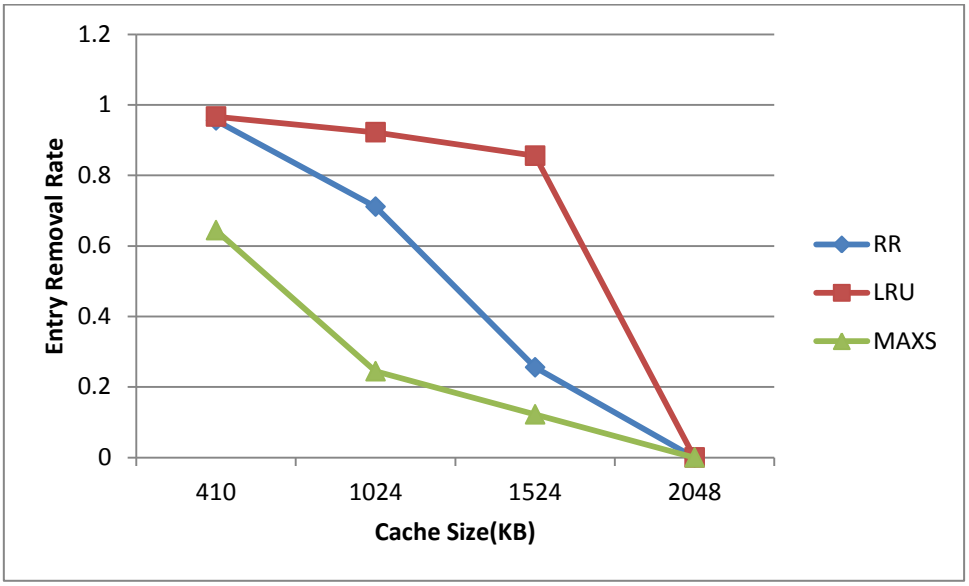

Figure 8: Entry Removal Rate vs. Cache Size

The results show that the entry removal rate is least for "MAXS" policy while "LRU" policy performs worst on the performance metric. The "RR" policy exhibits the performance closer to "LRU" for small cache sizes and closer to "MAXS" policy for large cache sizes.

Analysis

The results for this performance metric are quite expected. As discussed in section 7.1.3., "MAXS" policy is expected to show this behaviour as removes the largest page and thus, needs to remove less number of entries to provide space for a new entry. "LRU" policy displays poor performance because of the type of the workload. It has to remove cache entries for almost all the new requests until it gets sufficient cache size to keep most of responses for the requested URLs in the cache. Thus, graph shows that on an average, "LRU" policy has to evict almost a whole page for each new web page request that needs to be cached. "RR" shows a moderate performance indicating that given at least a moderate cache size, it can perform better. But, nothing can be concluded from its behaviour as it depends on the types of pages evicted at run time because of the random nature.

In case of cache size 2048 KB, all the three policies exhibit really good behaviour with the entry removal rate almost equal to zero. This is because of the fact that given the large cache, the contention is not there and almost all the requests can be cached.

7.2.4. Average Response Time

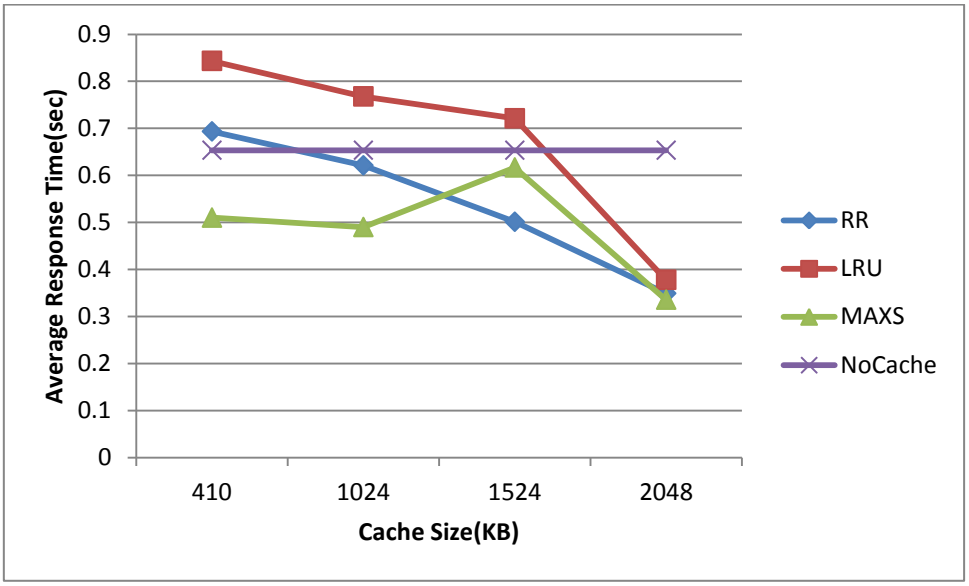

Figure 9: Average Response Time vs. Cache Size

The average response time for "No Cache" policy is around 0.65 seconds. "LRU" policy exhibits the average response time even worse than the "No Cache" policy. "MAXS" and "RR" policies show that the implementing a cache really improves the response time which is less than that of "No Cache" policy.

Analysis

"LRU" policy has a hit rate of zero until the cache size is set to 2048 KB. Because of this, the average response time of "LRU" policy is worse than the "No Cache" policy as it has to perform all the cache management operations which are just an overhead because keeping the cache is not fruitful. It can be concluded that under such workload, serving all the requests directly from the Web is a better option than implementing the cache with "LRU" replacement policy.

The "RR" policy performs worse than "No Cache" policy when the cache size is really small (410 KB) because the cache hit rate is zero in this case. But, it definitely performs better as the cache size increases. "MAXS" cache policy is good to implement even if the cache size is small as it evicts the largest page and thus, can serve more requests from the cache resulting in a better average response time.

Some variations have been observed in the case of "MAXS" policy when it exhibits the average response time near to that of "No Cache" policy with the cache size equal to 1524 KB. It may be due to the network response time and thus, can be ignored. On running the experiment again and again, we get different values for these performance metrics and thus, we can conclude that this may be due to delay from some external server.

Overall, we can conclude that the "MAXS" policy is the best option among these policies given such workload. We can't say anything about "RR" policy definitively but it would definitely perform better than the "LRU" policy for this particular type of workload.

7.3. “Anti-MAXS” Workload

This workload has been designed to exhibit the scenario in which “MAXS” policy would not be a good option for cache replacement policy. The workload consists of 90 URLs. First, sufficient URLs are requested to warm the cache (1024 KB size has been targeted) and two largest sized web pages are requested alternatively.

The result of “No Cache” policy is shown only in the case of average response time as it can be useful as a point of reference only in that case.

7.3.1. Cache Hit Rate

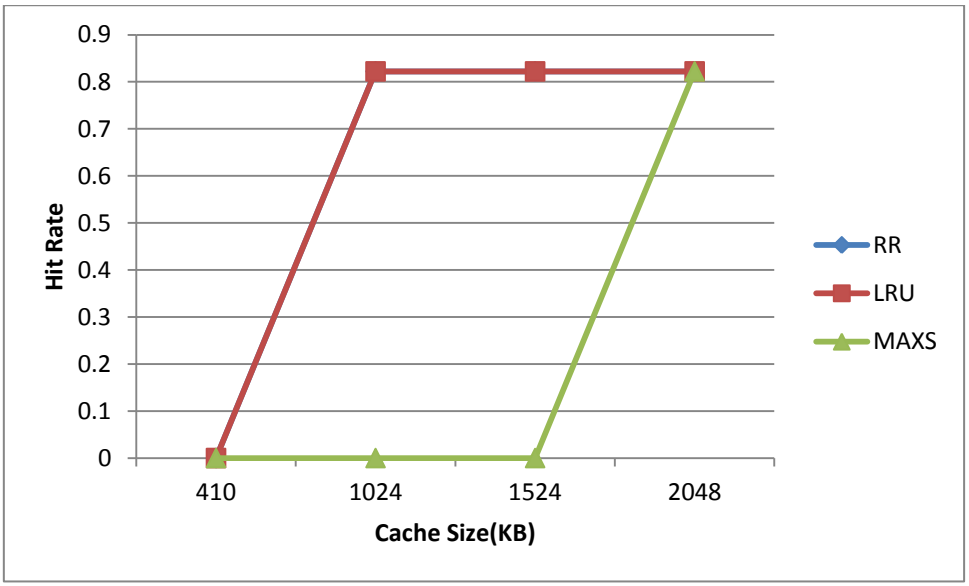

Figure 10: Cache Hit Rate vs. Cache Size

The results show that the “MAXS” policy performs worst under this workload and never gets a hit until the cache size is set to 2048 KB. The “RR” and “LRU” policies perform really well except the case when the case size is really low. All the policies perform equivalently when sufficiently large cache size is provided.

Analysis

The cache is first warmed by requesting a set of URLs consisting of a combination of URLs corresponding to both small and large sized web pages and then, the two largest web pages are requested alternatively. As the cache is full, on requesting a URL (which is not already in the cache and size larger than any entry in the cache), “MAXS” policy evicts the page with the largest size. Now, on requesting this evicted web page, it fetches the web page from the web and creates space in the cache by evicting the page which was inserted in the last request as it is the largest page. This behaviour continues on each request and thus, the requested web page is never found in the cache. This results in zero cache hit rate. For the cache size 2048 KB, sufficient space is available in the cache and thus, both the requested large web pages can be kept in the cache resulting in good cache hit rate.

Both “RR” and “LRU” policies performed equivalently. 410 KB is not sufficient to keep both the large pages together in the cache even if all the other entries have been removed. That is the reason why both of these policies exhibit cache hit rate zero. But, as the cache size is changed to 1024 KB, sufficient space is available to keep both the entries and thus, the cache hit rate shoots because the evicted page is either random or least recently used.

In this scenario, “RR” policy has performed same as “LRU” policy but it could have a lower cache hit rate in case the randomly chosen page for the eviction was the largest one. But, on an average, it would perform much better than “MAXS” especially if the randomness is uniform.

7.3.2. Byte Hit Rate

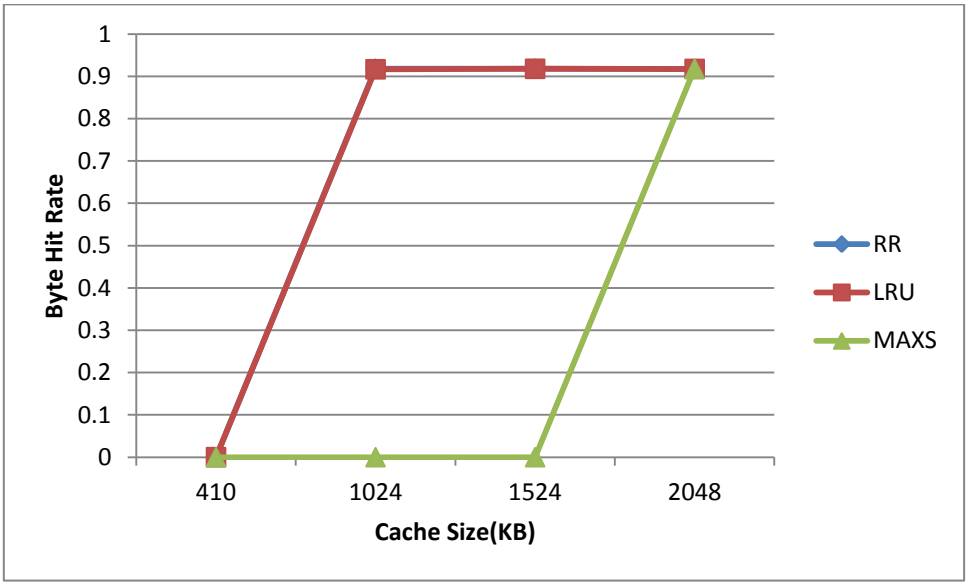

Figure 11: Byte Hit Rate vs. Cache Size

The results are almost similar to that of cache hit rate. The byte hit rate for “MAXS” policy is zero until the cache size is set to 2048 KB. Both “RR” and “LRU” policies perform well and have a byte hit rate more than 0.9 except the case when the cache size is 410 KB.

Analysis

The cache hit rate is zero for “MAXS” policy as the cache hit rate is zero till the cache size 1524 KB and thus, all the bytes are returned from the web only. When the cache size is set to 2048 KB, most of the requests are served from the cache except the initial requests and thus, the byte hit rate shoots up.

The cache hit rate for “RR” and “LRU” is zero for cache size 410 KB and thus, all the bytes are returned from the Web resulting in byte hit rate to be equal to zero. Once the cache size is set to 1024 KB or more, most of the requests except the initial requests are served from the cache and therefore, the both the policies produce good results for the byte hit rate.

7.3.3. Entry Removal Rate

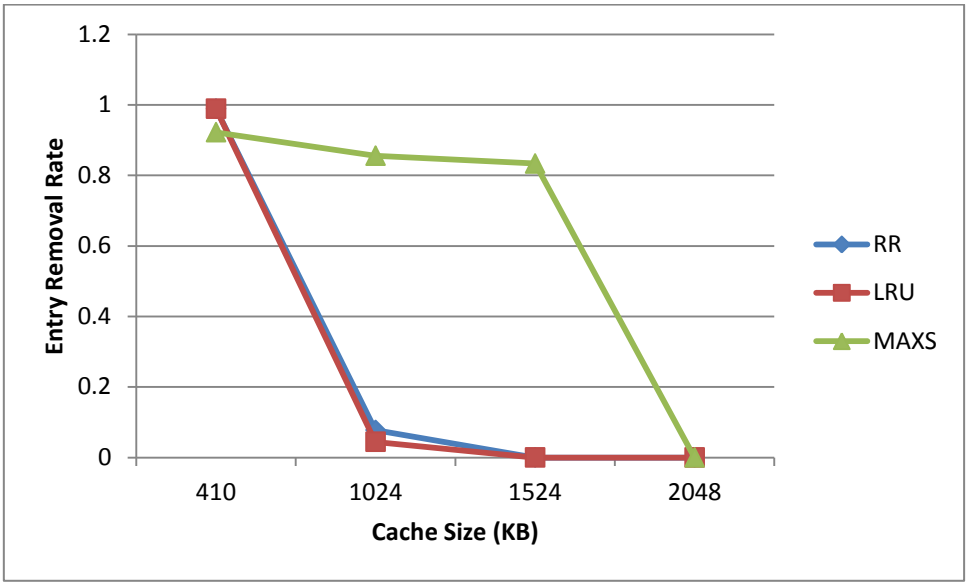

Figure 12: Entry Removal Rate vs. Cache Size

"MAXS" policy performs worst and has an eviction rate of almost one until the cache size becomes sufficiently large. "RR" and "LRU" policies exhibit almost same behaviour with entry removal rate equal to one for cache size 410 KB and it shoots up as the cache size is set to 1024 KB.

Analysis

"MAXS" policy performs worst as it has to evict page almost on each request after the cache gets filled once. When the cache size is set to 2048 KB, the "MAXS" policy gets sufficient space to keep both the web pages requested alternatively and thus, there is no need to evict the pages resulting in the cache eviction rate to be zero.

The "RR" and "LRU" policies have an entry removal rate of one for cache size 410 KB and the cache size in not sufficient to keep both the web pages together and thus, one page has to be evicted to create space for the other. For 1024 KB cache size, the eviction rate is really low as the targeted contention was designed keeping 1024 KB cache size in mind. Some pages might have to be removed to create space for the initial requests of these two pages once the cache is full. For the cache size 1524 KB or more, there is no need to evict any page as sufficient space is there in the cache to keep all the requested web pages and thus, the entry removal rate becomes zero.

Again, "RR" policy could have behaved differently depending upon the uniformity in the randomness.

7.3.4. Average Response Time

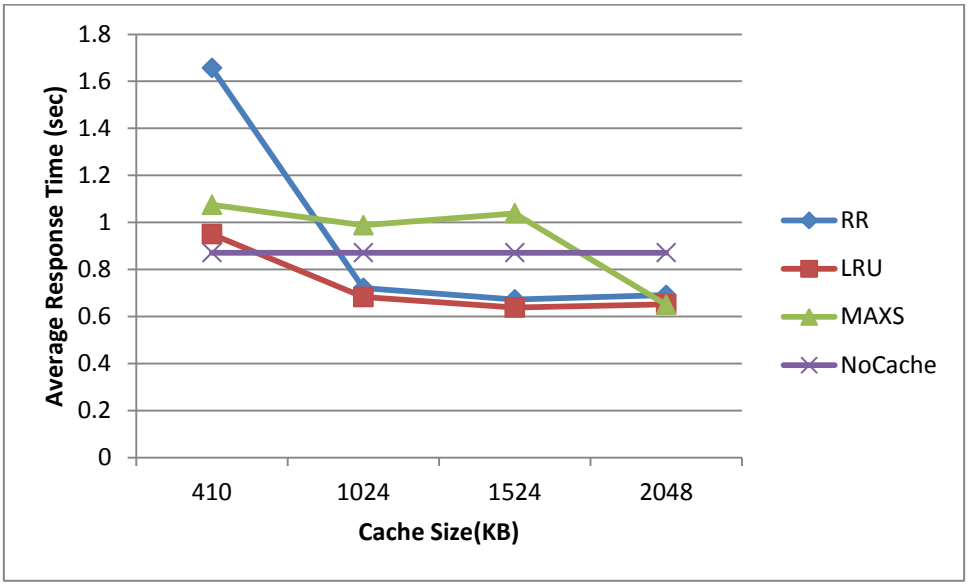

Figure 13: Average Response Time vs. Cache Size

The average response time for the "No Cache" policy is around 0.85 seconds to serve a request averaged over 90 URLs. The "MAXS" policy takes time more than the "No Cache" policy to respond the requests until the cache size is set to 2048 KB. Both "RR" and "LRU" policies have performance better than the "No Cache" policy except the case when the cache size is 410 KB.

Analysis

For the "MAXS" cache replacement policy, the cache hit rate is zero till the cache size 1524 KB and thus, almost all the requests are served from the Web. In addition to this task, the cache management operations are also carried which just become an overhead as we are not able to enjoy the benefit of the cache. Because of all this, "MAXS" policy performs worse than the "No Cache" policy which also serves the requests from the web but doesn't have to perform cache management operations. "MAXS" performs better only when it has been provided sufficient space to keep the web pages. Therefore, under this workload, the "MAXS" policy would not be good choice if the cache size is limited.

Both "RR" and "LRU" policies exhibit the cache hit rate equal to zero when the cache size is 410 KB. Because of this, the web pages have to be fetched from the web and thus, the average response time is worse than the "No Cache" policy because of the additional cache management overhead. Once the cache size is set to 1024 KB or more, the cache hit rate increases and thus, fewer web pages are required to be fetched from the web. Therefore, the response time decreases and both the policies perform better than the "No Cache" policy.

The "RR" policy shows really high average response time for cache size 410 KB and this can be attributed to the variations in the network delay and the response from the external servers.

7.4. “Most Common” Workload

This workload has been designed on a complete experimental basis. It consists of 90 URLs and a mix of frequently and less frequently accessed URLs as well as URLs corresponding to both small and large sized pages. This is a pattern commonly observed in the daily based web access by a user.

The results of “No Cache” policy are shown only for the “Average Response Time”.

7.4.1. Cache Hit Rate

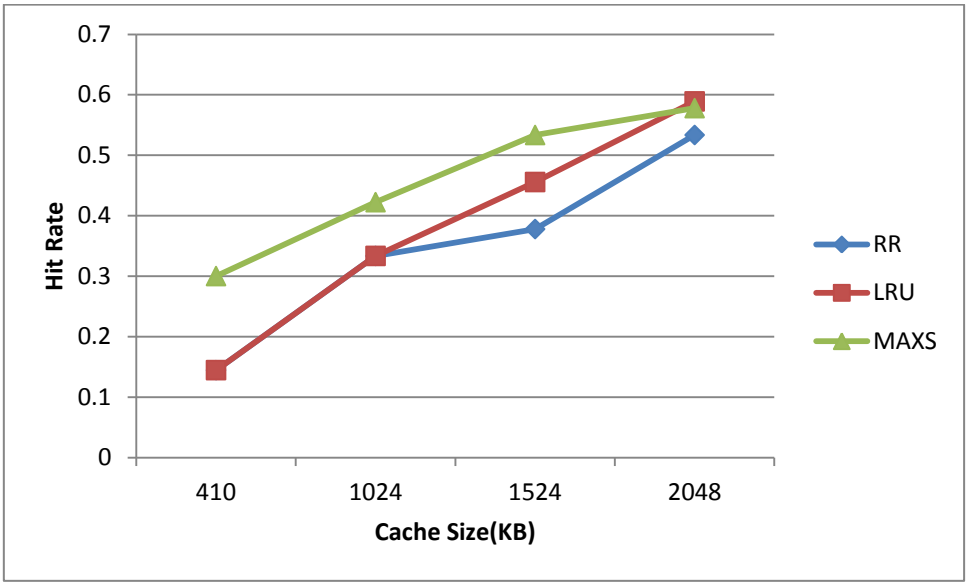

Figure 14: Cache Hit Rate vs. Cache Size

The “MAXS” policy shows best performance in presence of contention followed by “LRU”. The “RR” policy performs well but not in comparison to the other two policies.

Analysis

The “MAXS” policy performs better especially when the cache size is less. This is because of the fact that “MAXS” policy evicts largest page first thus, more number of different entries can be kept and served from the cache. As the cache size increases, “LRU” policy starts approaching the “MAXS” policy as role of cache size reduces and it more depends upon the access pattern. As the workload is a mix, both “MAXS” and “LRU” should perform well given moderate cache size.

Nothing concrete can be concluded from the behaviour of “RR” policy but, it would not be a better choice as it doesn’t take the benefit of the access pattern. It can be a good choice in a scenario where the access pattern is not known and have quite variations. Therefore, it can be concluded that “MAXS” would perform better given the cache size is small and both “LRU” and “MAXS” policies can be good options depending upon the access pattern. If a set of pages are accessed frequently “LRU” may even perform better than “MAXS” policy given at least some moderate cache size.

7.4.2. Byte Hit Rate

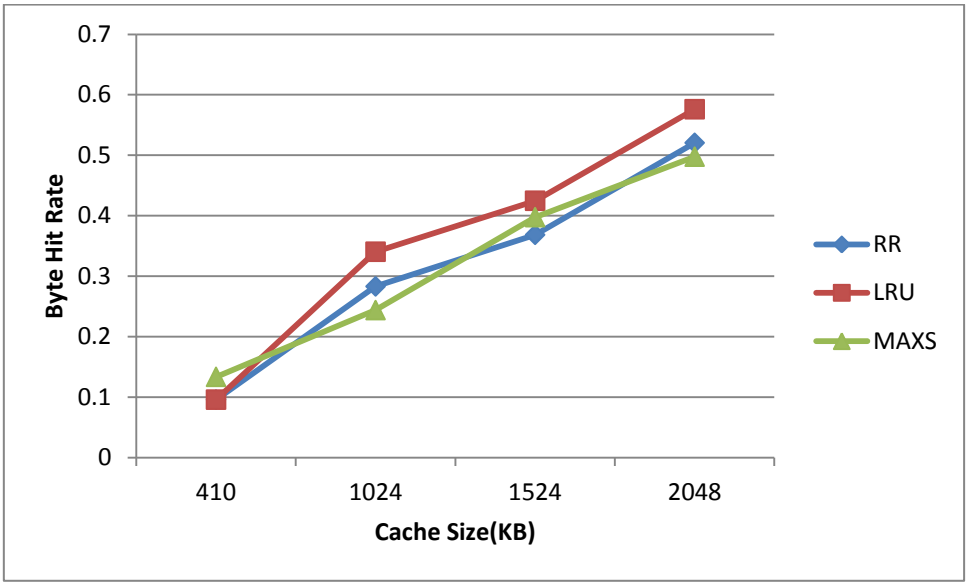

Figure 15: Byte Hit Rate vs. Cache Size

"LRU" performs best except the case when the cache size is small. "MAXS" and "RR" policies show an intermixed behaviour.

Analysis

For the cache size 410 KB, "MAXS" policy performs better as its cache hit rate is much higher than others and thus, it returns more number of bytes from the cache. But, as the cache size increases, "LRU" policy starts performing better. The reason behind this is the fact that "LRU" evicts the pages on basis of usage time while "MAXS" evicts the pages on basis of page size. So, if there are requests for large pages, proxy server with "LRU" policy implemented can serve this request from the cache more often whereas it would have to fetch the pages from web in case of "MAXS" as it is high likely that the large sized pages were evicted from the cache.

"RR" and "MAXS" policies show intermixed behaviour and nothing concrete can be concluded from this. This is because "RR" policy evicts pages randomly and might have evicted large sized pages sometimes while small sized pages other times. In another run, "RR" policy may even perform better than the "MAXS" policy.

So, the key point to be noted is that for the byte hit rate performance metric, "LRU" would be a good and reliable replacement policy.

7.4.3. Entry Removal Rate

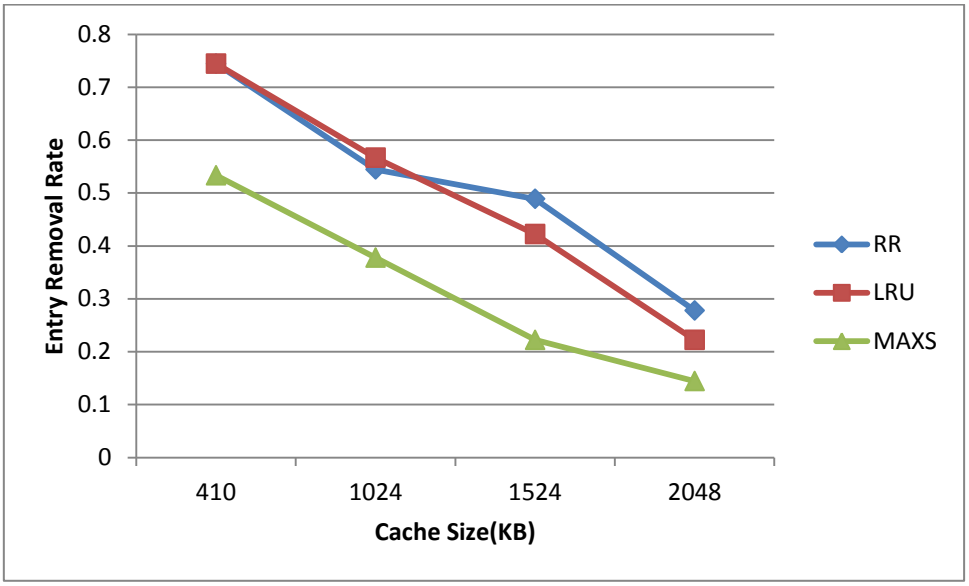

Figure 16: Entry Removal Rate vs. Cache Size

"MAXS" policy shows the best performance for this metric. "RR" and "LRU" policies have the same behaviour till the cache size 1024 KB and after that, "LRU" policy performs better than "RR" policy.

Analysis

The result for the "MAXS" policy is quite intuitive. As the largest page gets evicted and thus, less evictions are required to create space for a new entry. "RR" and "LRU" perform almost similar until the cache size is comparable to the contention. For larger cache sizes, LRU performs better. It may be because of the fact that the workload has a pattern of most frequently accessed URLs and the cache hit rate is high. Therefore, given reasonable cache size, the server with "LRU" policy implementation can serve more requests from the cache and would have to evict less number of pages.

It is not possible to conclude anything concrete about the behaviour of the "RR" policy because of its random nature. But, on an average, it can be concluded that "MAXS" policy would perform best followed by "LRU" for this particular performance metric.

7.4.4. Average Response Time

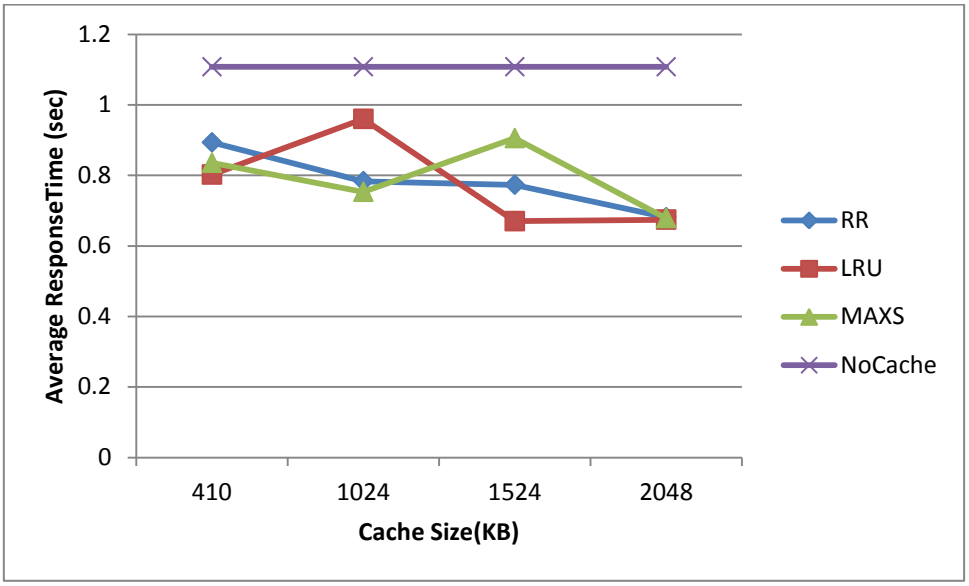

Figure 17: Average Response Time vs. Cache Size

The "No Cache" policy takes, on an average, around 1.1 seconds to serve a request. All three implemented policies exhibit intermixed behaviour but definitely perform better than the "No Cache" policy.

Analysis

This result matches our hypothesis and shows that in practical scenarios, implementing a cache is always good and it saves much transmission time providing the response to the clients at faster speed.

All the three policies have better response time than "No Cache" policy. We can't conclude anything for these three as they show some variable behaviour and "RR" exhibiting a moderate performance among these three. These variations are due to the network delay and the variation in the response from the external web servers. In another run, these three may produce different results but would definitely perform better than the "No Cache" policy and that is the most important take away.

Hence, we can conclude that under a practical workload, "MAXS" policy may be a good choice in case the caches are small. But, given a reasonable cache size, both "MAXS" and "LRU" may be good contenders for the replacement policy and would depend upon the expected usage pattern. If the network communication is costly and delay is expected in the network, "LRU" policy may be a better choice as it results in highest byte hit rate.

8. Conclusion

In this project a Web Proxy Server supported by remote procedure calls has been implemented. Apache Thrift technology has been used to provide RPC mechanism. We build a Web Page cache for the Proxy Server with different cache replacement policies. In our experiments we analyze the performance of different cache policies by measuring their performance as per the different metrics for a set of workloads with different cache sizes. The workloads are designed specifically to ensure that different policies work optimally for some particular workload.

From the results, we can conclude that for small sized caches MAXS replacement policy performs better as it removes the largest page which leads to lesser evictions in the already small sized cache. Because of this, more number of entries can be kept in the cache resulting in higher cache hit rate.

As the size of the cache increases both LRU and MAXS approach a similar performance. As the size of the cache increases further the performance of the cache depends heavily on the pattern of the workload for both LRU and MAXS policies.

Two of the workloads, namely “anti-MAXS” and “anti-LRU” are designed specifically to perform worse over the other in a specific workload pattern. The third workload namely “most-common” is closer to reality. Analysis of the effect of the cache replacement policies on this workload suggests that the performance heavily depends on the properties of the URL s in the workload i.e. size of the pages associated with that URL and also on the pattern in which the URLs are accessed. For instance if large sized URLs are accessed frequently MAXS would give a worse performance over LRU. Similarly, LRU may perform worse in cases where the workload resembles a round robin access pattern.

To conclude if we take a scenario where we have a descent cache size and the access pattern is not heavily biased, LRU cache replacement policy should give the most optimal results. Also if the network latency is high or network bandwidth crucial, LRU performs better since Byte Hit Rate is higher for LRU policy.