RPC-Based Web Proxy Server: Implementation and Evaluation using Sun RPC and libcurl

Machiry Aravind Kumar Prabhavathy Viswanathan

Georgia Institute of Technology
{amachiry,pviswanathan6}@gatech.edu

Abstract

In computer networks, a proxy server is a server (a computer system or an application) that acts as an intermediary for requests from clients seeking resources from other servers. A client connects to the proxy server, requesting some service, such as a file, connection, web page, or other resource available from a different server and the proxy server evaluates the request as a way to simplify and control its complexity. In this project we have implemented web proxy, which facilitates access to content on the internet. The proxy server exposes a Remote Procedure Call interface through which client(s) can request URLs(web pages). Caching of web pages is enabled on server and five cache replacement policies are implemented to manage the cache. We present the details of our experimental data and the methods we used to obtain it along with detailed evaluation of the performance of replacement policies.

1 Introduction

A proxy server is a server (a computer system or an application) that acts as an intermediary for requests from clients seeking resources from other servers. A client connects to the proxy server, requesting some service, such as a file, connection, web page, or other resource available from a different server and the proxy server evaluates the request as a way to simplify and control its complexity. There are different types of proxy servers depending on the supported content like web proxy, file proxy etc. In this project, we concentrate on web proxy server where the server serves web pages.

A caching web proxy server accelerates web page requests by retrieving content saved from a previous request made by the same client or even other clients. Caching proxies keep local copies of frequently requested resources, allowing large organizations to significantly reduce their upstream bandwidth usage and costs, while significantly increasing performance. Most ISPs and large businesses have a caching proxy. Caching proxies were the first kind of proxy server. Another important use of the proxy server is to reduce the hardware cost. An organization may have many systems on the same network or under control of a single server, prohibiting the possibility of an individual connection to the Internet for each system. In such a case, the individual systems can be connected to one proxy server, and the proxy server connected to the main server. Due to the limited cache space, replacement of the cache entries is inevitable when there are new resource requests. There are various well known replacement polices which pick the cache entry to be replaced based on a criteria.

In this project, we develop a caching web proxy server which exposes RPC interface to query the contents of web pages. We also implement five replacement polices and evaluate their performance on 2 **real datasets**. We use Sun RPC to implement the RPC interface and libcurl to fetch the web pages.

Sun RPC is a widely-used protocol for RPC in different programming languages. It relies on an available service (RPC portmap) to handle binding and service location. ONC RPC also relies on the XDR standard (eXternal Data Representation) to define a common wire-format for structured data sent between machines. It also defines an interface definition language that can be used with rpcgen to automatically generate stub

Task	Owner
Interface and data structure design	Both at a meeting
Server Design	Individual research and decided at meeting
Server Implementation	Machiry
5 Replacement algorithms	Prabha
Test Harness and Performance Measurements	Machiry
Write up	Prabha and Machiry

Table 1: Division of work

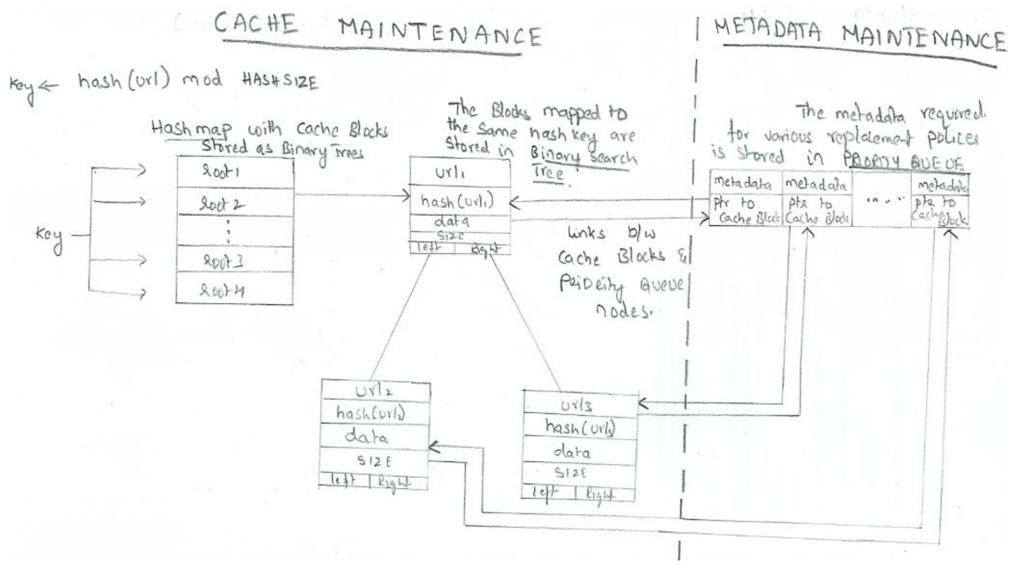

CACHE MAINTENANCE
METADATA MAINTENANCE
Key ← hash (url) mod HASHSIZE
Hashmap with cache Blocks stored as Binary Trees
Key
Root1
Root 2
Root 3
Root 4
The Blocks mapped to the same hash key are stored in Binary Search Tree.
url₁
hash(url₁)
data
size
left
Right
url₂
hash(url₂)
data
SIZE
url₃
hash(url₃)
links b/w Cache Blocks & Priority Queue nodes.
The metadata required for various replacement polices is stored in PRIORITY QUEUE.
metadata
ptr to Cache Block

Figure 1: Data structures

code for services (as well as marshalling and unmarshalling code). 'libcurl' is a powerful library for communicating with servers via HTTP (and FTP, LDAP, HTTPS, etc.). It supports HTTP GET/PUT/POST, form fields, cookies, etc.

The remainder of the report is organized in sections as follows : division of work, implementation details of the interface, including, design and various data structures used, experimental set up and evaluation, and Conclusion.

2 Division of work

We set up an initial meeting to decide on the RPC interface and the data structures to be used for efficient access. Then, we did our individual research on efficient server design so as to make it generic so that addition of any replacement policy should be a trivial task and decided on a design. We held another meeting to discuss the replacement polices and obtaining experimental data. We concluded to use real data from chrome history files for our evaluation. The detailed split up of the tasks are shown in Table 1.

3 Implementation

As mentioned previously, we use SUN RPC to implement our RPC interface. Our interface definition is as shown below:

```
struct webpageResponse {
    string webPageContents<20971520>; //Max Size : 20 MB
};
typedef struct webpageResponse webpageResponse;

program PROXY_FETCH_PROG {
        version PROXY_FETCH_VERS {
                webpageResponse PROXY_FETCH(string) = 1;
        } = 1;
} = 0x23451111;
```

The interface contains only one method: PROXY_FETCH which accepts one string parameter and returns a webpageRepose structure containing the contents of the requested web page in a null terminated string. We generated multi-threaded implementation using -M option for rpcgen to generate the server code. Although we have cache size implemented as both number of urls and total storage size, **we consider cache size to represent: maximum number of urls whose contents (irrespective of their size) can be stored and use this for our evaluation**.

3.1 Data Structures

In order to optimize the cache look up and selecting the cache entry to be replaced, we use specific data structures as explained below. The organization and the usage of the various data structures is as shown in Figure 1

3.1.1 Cache HashTable/HashMap

We use a Hash Table to store the pointers to the structures which contain the contents of urls. The hash table is stored as an array and its zero indexed. The formula to compute the key for a URL is: $key = djbhash(url) \, mod \, HASHSIZE$ where $HASHSIZE$ is the maximum number of entries that can be stored in the hash table and *djbhash* [2] is the Hash function provided by Dan Bernstein. 'hash' and '*djbhash*' have been used interchangeably in the report unless explicitly mentioned otherwise.

3.1.2 Binary search trees to resolve collisions/aliasing

In order to handle collisions when *key* of more than one url is same, we store the following data for each url being fetched in a C structure: url, $djbhash(url)$, size, contents and a link to the structure containing metadata (required for replacement policies). All the nodes having the same *key* are organized as binary search tree [3] with search key being the hash of the url. When the contents of a url is requested, we compute the *key* for the hash table as explained before and hash of the url. We fetch the corresponding pointer from the hash table by using the *key*, using binary search tree semantics we find the node containing the contents of the url using the hash. To ensure correctness, for each node having same hash value, we also do a string comparison of the url. This is an inspiration from the page sharing technique used in VMWare VMX Server.

3.1.3 Priority Queue or Heap for metadata maintenance

Most of the replacement polices require some form of metadata to be maintained for each cache block and the metadata needs to be organized in some priority so that finding the next candidate to replace would be easy. For example: in case of LRU, the metadata is a timestamp of the last access and we need to organize this data such that the earliest timestamp can be easily accessed. Using this observation, we maintain the metadata for each cache block in a separate node called metadata node. Each cache block has a link to the corresponding metadata node and vice versa. All the metadata nodes are organized in a priority queue by using the interface specific to the current policy.

3.2 Design

We used interface based design. Our base implementation is a Hash table with binary search trees for maintaining cache blocks and a priority queue for maintaining metadata nodes. The priority queue needs an implementation of an interface to organize the queue. As we use the language C for our implementation,

Replacement Policy	**Metadata type**	**Updating metadata**	**Ordering of priority queue**
LRU	struct timeval	we update every time the cache block is accessed with current time stamp	increasing order of the timestamp
LFU	int	we increment the count for every access	increasing order of the count value
FIFO	long	we assign the current index value at the time of first access	increasing order of the index value
RAND	int	a random integer at the time of first access	increasing order of the assigned value
MAXS	long	size of the fetched webpage at the time of first access	decreasing order of the size value

Table 2: Type of Metadata and ordering of the priority queue based on replacement policy

the interface is in the form of function pointers. Any new replacement policy just needs to implement the interface and the corresponding implementation can be used to organize the priority queue. The interface that need to be implemented by any replacement policy is as shown below:

```
/*
 * This is to compare 2 metadata nodes , return -1 if
 * metadata of arg1 < arg2 , 0 if same and 1 if
 * arg2 > arg1
 */
typedef int (*CompareHeapNodes)(HeapBlockPtr,HeapBlockPtr);

/*
 * Link the cache block with the metedata node and
 * initialize the metadata value
 */
typedef void (*LinkAndInitialize)(CacheBlockPtr,HeapBlockPtr);

/*
 * update the metadata of the provided cacheblock
 * This is called every time this cacheblock is fetched
 */
typedef void (*UpdateMetaData)(CacheBlockPtr);

/*
 * create a metadata node
 */
typedef HeapBlockPtr (*GetNewHeapNode)(void);
```

The source code structure is shown in Figure 2

We implemented 5 replacement policies:**LRU, LFU, FIFO, RAND and MAXS**. Table 2 shows the the type of metadata maintained and ordering of the priority queue.

The runtime of various operations is as shown in Figure 3, where n is the cache size.

3.3 Implementing new Replacement policy

Implementing a new replacement policy is straightforward. It requires implementing the functions mentioned before as adding a case in the initialization routine (*initializeGlobalData*) in the proxy_server.c file.

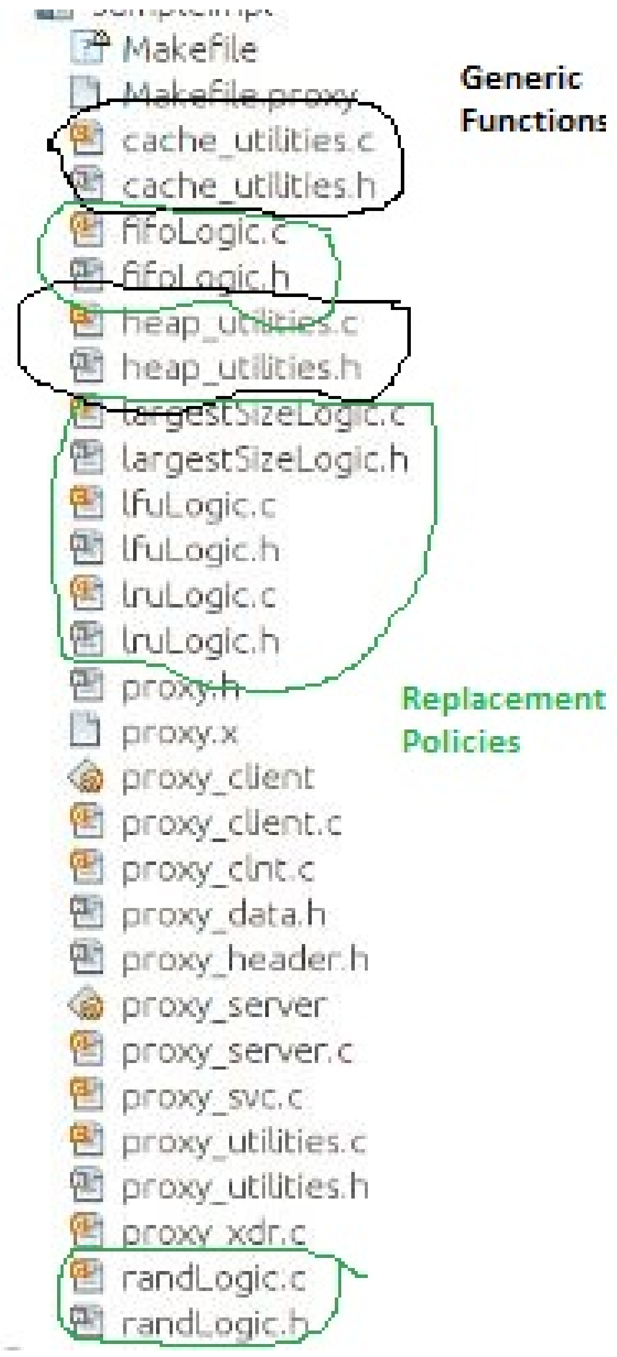
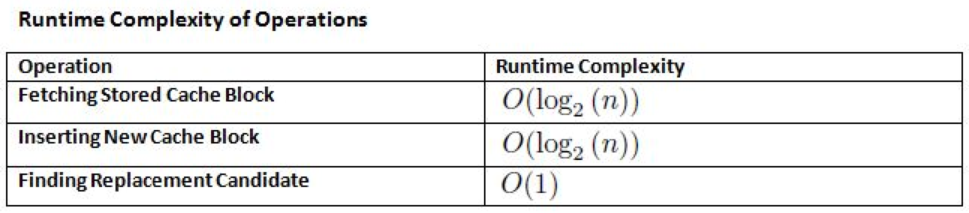


Figure 2: Source code structure



Runtime Complexity of Operations

Operation	Runtime Complexity
Fetching Stored Cache Block	$O(\log_2(n))$
Inserting New Cache Block	$O(\log_2(n))$
Finding Replacement Candidate	$O(1)$

Figure 3: Run time complexity

4 Evaluation

We did an extensive evaluation of our implementation using two different datasets gathered from web access traces of two different users. For each dataset and each replacement policy, we computed the runtime, hit rate and run time behaviour of the replacement policies for cache sizes varying from : 10% to 50%(in steps of 10) of distinct URL's in the dataset, and one at 80% to show the effect of cache size on various performance metrics. We first describe how we got the experimental data and the characteristics of it, followed by a detailed evaluation of the various replacement policies on the data.

4.1 Experimental Data and set up

We first tried to do sampling of the the 100 best urls to generate our dataset, but we realized that this might bias our results as results can be predicted based on the distribution of data. In order to get real data, we decided to refer the browsing history of real users. We consulted two of our friends who were ready to share their browsing history. However, getting the browsing history manually might be cumbersome and time consuming. The 2 users from whom we requested to share the history data were using Google Chrome browser[4]. Google Chrome stores the history data[5] in sqlite files that are stored as part of google chrome config data in the home directory of the user.

We parsed the history files to get the browsing data for approximately 2 months. String comparison of all the urls showed that most of the urls were unique mainly because of the http get parameters (that are passed as part of request url), most of which are transient like session id, etc. To overcome this problem, we collected only base urls of all the urls present in the history files. considerable number of accessed urls that were on https, required redirection. Hence, we enabled https and allowed to follow redirection in our libcurl configuration.

The size of the dataset and the number of unique urls (required to adjust cache size) are shown in Table 3. Having less than 10% of unique urls in the url access emphasizes the need for cache based webproxy server and good replacement policy.

We ran our experiments with server running on Ubuntu 12.04 64-bit having Intel(R) Core(TM) i7-3610QM CPU 2.30GHz with 12 GB of RAM connected to Internet using Ethernet with download speed of 830 Mbps and upload speed of 132 Mbps , client running on Ubuntu 12.04 64-bit having Intel(R) Core(TM) i7-3610QM CPU 2.30GHz with 4 GB of RAM connected to server through wired LAN.
The access patterns of data sets is shown in Figure 4 . In this figure, the x-axis is the id of the url and y-axis is the individual url access. From the chart it is quite evident that in case of dataset 1 majority of the access are to a single url (you can see a single line dominating the chart). In fact 44.7% access are to a single url. Also, most of the access are to the url which are accessed initially. Unique urls that are accessed later are accessed relatively less number of times. Whereas in case of dataset 2, the accesses are more uniform with lots of horizontal lines well distributed. In short, **horizontal lines represent temporal locality and the space between the horizontal lines represent spatial locality**. Therefore, dataset1 is having extremly high temporal locality and less spatial locality where as dataset2 is having both spatial and temporal locality. We also observe a frontier at the top. This indicates that when we see access to new url, we call this **New Url Frontier(NUF)**
The distribution of number of accesses for each url of the dataset is as shown in Figure 5. In this figure, the area of sector is indicative of the number of times the url is being accessed in the dataset. dataset1 is dominated by a single url with 44.7% of times being accessed. On the other hand, dataset2 is uniformly distributed.
The CDF(Cumulative Distributive Function) for sizes of web pages against the access percentage is shown in Figure 6. In this figure, the x-axis is the percentage of url accesses and y-axis is the size of the corresponding webpage. In case of dataset1, more than 75% of the requested urls are within : 20KB with 44% of the urls of size: 20KB. Similarly in case of dataset2, more than 62% of the urls are within : 20KB. This shows that most of the urls that users access are of less size.

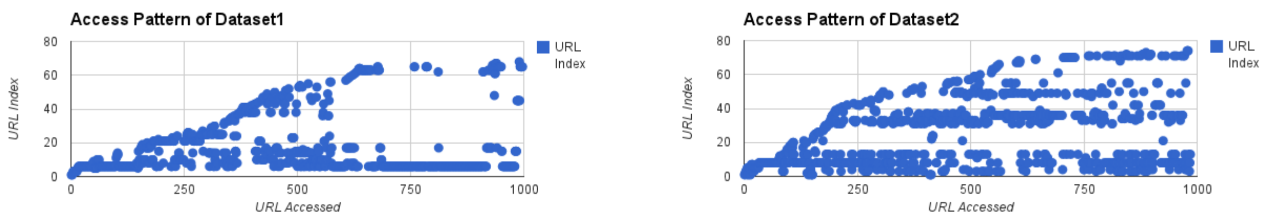


Figure 4: Data access pattern

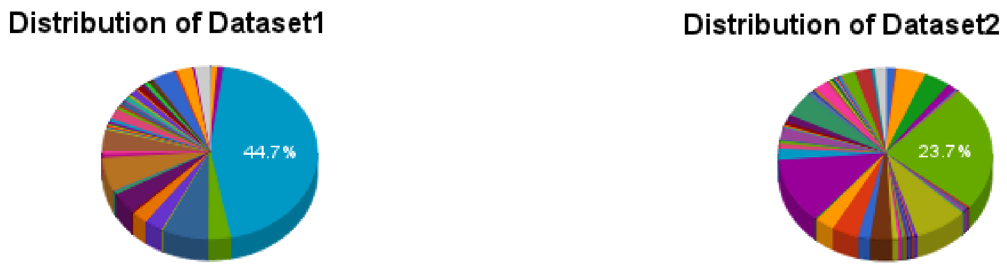


Figure 5: Number of accesses

4.2 Performance analysis of replacement policies

We measure performance of the replacement policies in three dimensions: by total time taken to access the complete dataset, cache hit rate and run time behaviour. We varied the cache size to measure the effect of cache on the various performance metrics. We also disabled cache completely and measured the total access time.

4.2.1 Access time

In this metric, we measure the total time taken to fetch the complete dataset by varying the cache size across different replacement policies. When cache is disabled the total access time is : 707.35s and 633.18s for dataset1 and dataset2 respectively which is 4 times the largest time taken and 2 times the largest time taken by any replacement policy with least caching for corresponding datasets. The time measured is the total time after the client makes the RPC call and before the response is received by the client. Figure 7 shows the access time of various replacement policies. The important observations that can be made from the chart are as follows:

- **As the cache size increases**
  - Access time decreases for all replacement policies.
  - The difference between the time taken by different replacement policies decreases. In fact, the difference is in the form of a parabola. It increases rapidly and reaches maximum very early at 10% and decreases slowly. It reaches zero when the cache size is 100% and when cache size is minimum i.e 1 entry.

Dataset	Size	No of unique urls
dataset1	997	68
datset2	985	74

Table 3: Dataset dimensions

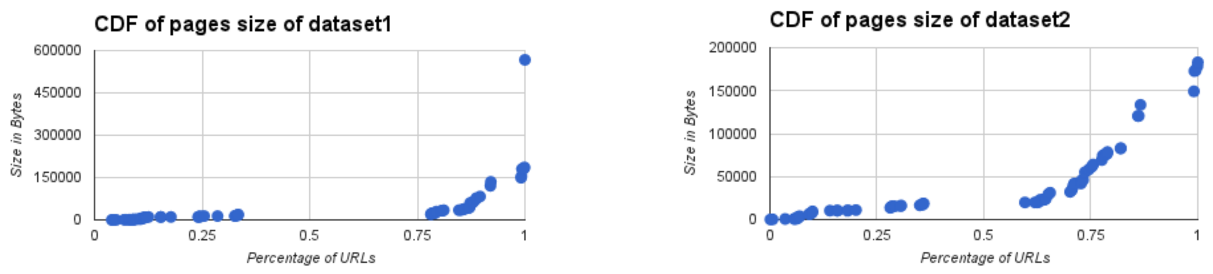


Figure 6: Webpage size CDF

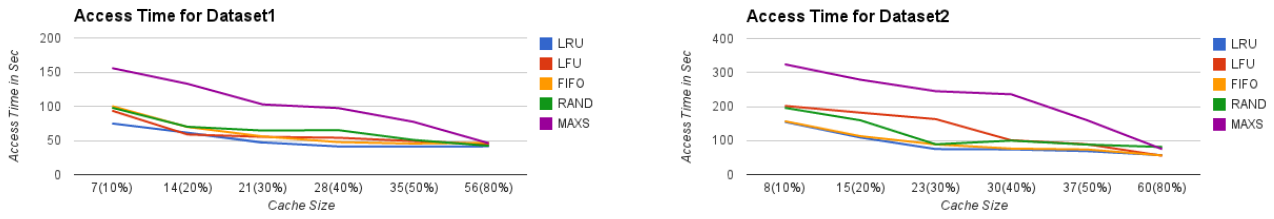


Figure 7: Access Time

  - For dataset1, no significant gain in time is achieved as we increase the cache size after 20%. This is expected from the access pattern in Figure 4, where only a small set of urls are frequently accessed (which is also evident from the empty space towards the right of the chart).
  - Similarly for dataset2, no significant gain in time is achieved after cache size is 30%. This is because the actively accessed urls are analogous to working set in processor cache is 30%, which is also evident from the several horizontal lines of increasing height as we move right in Figure 4.

- For dataset1, LRU performs best in all cache sizes, followed by LFU, FIFO, RAND and MAXS in that order, with RAND showing some discrepancies as expected. This can be expected from the access pattern of dataset1. LRU performs best as the urls are accessed in clusters (horizontal lines in Figure 4). LFU is the next best as the distribution of urls is not uniform with only few urls being accessed many times as indicated by Figure 5. FIFO comes next as the working set is small. RAND is again random as expected. MAXS performs the worst, because from Figure 6 more than 75% of the requested urls are within : 20KB with 44% of the urls of size: 20KB. Therefore, the probability of picking the most accessed url is maximum. This results in a ping-pong effect with the most accessed url cache block being flushed and stored repeatedly.

- Similarly for dataset2, LRU performs best in all cache sizes, followed by FIFO, LFU, RAND and MAXS in that order. The main difference is that LFU takes more time than FIFO. This can be attributed to the uniform distribution of the accessed urls, from Figure 5. It can be easily observed that majority of the urls have been accessed same number of times and the access to them is uniformly distributed. MAXS again performs worst because of the same reason explained above for dataset1. We also observe a subtle difference here. MAXS performs relatively better in comparison with the MAXS performance on dataset1, which again can be attributed to the access pattern (Figure 4) and distribution (Figure 5).

4.2.2 Hit Rate

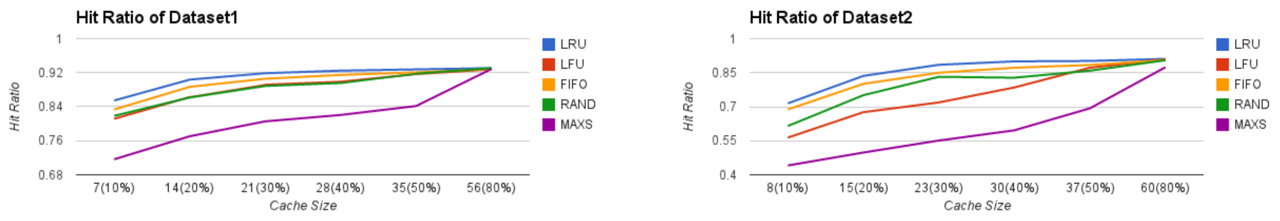


Figure 8: Hit Rate

In this section, we evaluate the cache hit rate of different replacement polices as we vary the cache size. Figure 8 shows the results for 2 datasets. As expected, the lines in the chart are almost inverse of those in Figure 7 emphasizing the fact that access time is inversely proportional to the hit rate.
Also, it is important to note that for MAXS replacement policy, hit rate increases smoothly whereas access time (Figure 7) decreases more rapidly. This can be attributed to the web access time and network time. When hit rate is less, we end up fetching large web pages more often which directly contribute to the access time.

4.2.3 Run time behaviour

In this section, we evaluate the behaviour of each replacement policy as we increase the cache size. The behaviours we are interested are hits and misses. In the Figure 9

- We indicate the position in the dataset when we observe the first cache miss (This is indicated by a line from 0). This also shows when the cache is filled. We call it **Initial Cache Miss Line (ICML)**
- x-axis indicates the position of the url in the dataset and on y-axis, 2 indicates cache hit and 1 indicates cache miss.
- The color pattern corresponds to the access pattern in Figure 4, with horizontal lines representing empty space as they will always result in cache hit.

For FIFO and RAND, we noticed a situation where a cache block which was hit when cache size is less was missed when we increased the cache size. After a bit of research, we learned that this is because of **belady's anomaly** [6].
The following observations can be made from the Figure 9

- The chart of LRU shows a lot of empty spaces indicating more hit rate and the effectiveness of the replacement policy.
- The ICMLs are more closely spaced in dataset2 than in dataset1. This can be attributed to the access pattern of the datasets as shown in the Figure 4 and the corresponding NUFs. Initial slope of NUF is more for dataset2 which indicates new urls are seen quickly resulting in cache getting filled up quickly and cache misses start appearing early.
- RAND for dataset2 shows sudden empty space indicative of its random nature.
- ICMLs are variably spaced for the same amount of increase in cache. This again can be attributed to the NUFs of the datasets. The space between ICMLs is directly proportional to the slope of the NUFs for the corresponding increase in cache size.

5 Conclusion

Real web access are largely skewed emphasizing the need of a cache based web proxy. Most of the web sites accessed are of less size (20 KB average size) reducing the demand on the cache size. LRU is the most effective replacement policy irrespective of the access pattern. Effectiveness of LFU largely depends on the data access pattern. FIFO replacement policy is easy to implement and results in decent performance. MAXS leads to the worst performance when cache size is maintained as number of urls and majority of the urls are of same size.

References

[1] *Wikipedia.* http://en.wikipedia.org/wiki/Proxy_server

[2] *DJBHash.* http://www.cse.yorku.ca/ oz/hash.html

[3] *Binary Search Tree.* http://en.wikipedia.org/wiki/Binary_search_tree

[4] *Google Chrome.* https://www.google.com/intl/en/chrome/browser/

[5] *Google Chrome History File.* http://productforums.google.com/forum/!topic/chrome/6ndeqflOq0M

[6] *Belady's anomaly* http://en.wikipedia.org/wiki/Beladys_anomaly

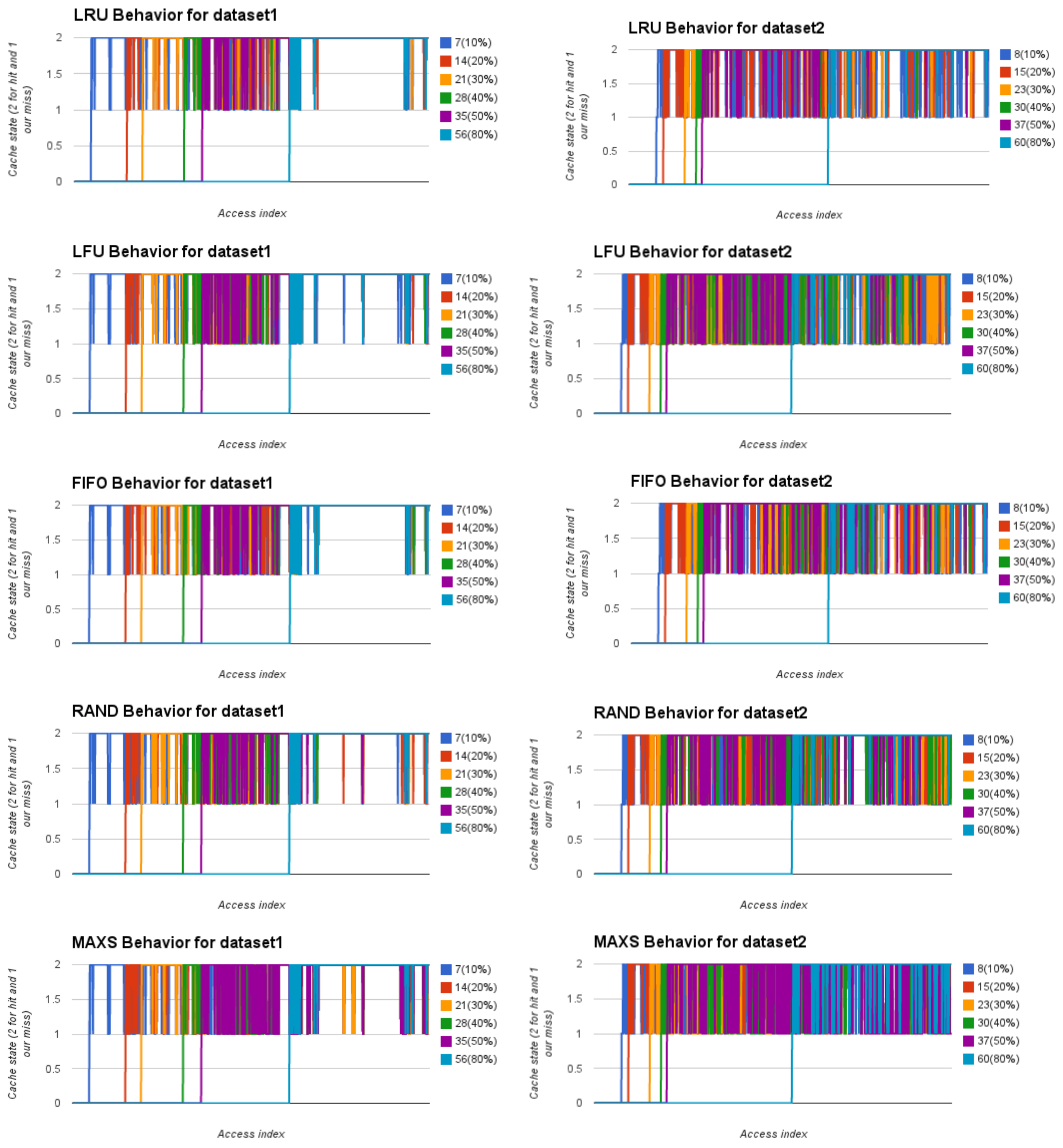


Figure 9: Behaviour of various Replacement policies