PROJECT 3
RPC-Based Proxy Server

Girish Dhoble
Swapnil Deshpande

Introduction

A web proxy server is a server 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, 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. A proxy server can store the web page in its cache so that for future request for the same page, the proxy server that send the cached page to client instead of getting it from the original server, thus improving the performance. The size of cache should be such that it should be fast to access the cache and also the number of cache hits should be high. Various cache replacement policies are used to evict cache pages when there is a new client request and the cache is full. We developed a web proxy server using three cache replacement policies: Random policy, FIFO policy and MAXS policy. We used metrics such as cache hit ratio and service time (Latency) to determine the effectiveness of our cache replacement policies. We created 4 workloads for testing our cache replacement policies and also varied the cache size from 1MB to 5MB. We found out that random policy performs moderately across different workloads, FIFO performs the best when the URL patterns exhibit locality and MAXS policy works the best when the dataset contains some large size pages which are rarely accessed by the user again due to its size. But MAXS leads to the worst performance when the majority of the URLs are of same size.

Cache Design Description:

We used a map to be the data structure for the cache. The reason behind using a map is that it is easy to lookup the cache by a key. The map provided by the C++'s Standard Template Library (stl::map) was used. We used a map because its really easy to search a value in a map using a key and searching for a previously fetched body (the HTML page without the header) of a web page from a URL is really a default way to search that comes to the mind. Of course we don't argue that there couldn't be other ways that could be implemented apart from our own. In fact, there could be some better implementations but this is what we thought would be a good idea.

The key-value pair used in the map are, both, strings (i.e. map<string, string> is used). This map is actually defined in the thrift file in a struct called `MyMap`. The name if the map is `URLBody` (since it maps URL to its Body). The reason for defining the map in the struct is Thrift only allows constant variables at a global scale.

URL(key)	**Body**
www.google.com	<html>...
...	...
...	...

Fig: URLMap having URL as the key and the HTML page (without header) as the value

The map is internally implemented by C++ as self-balancing binary tree and its complexities are as follows:

Searching: O(log n)
Insertion: O(log n)
Deletion: O(log n)

Caching Replacement Policies:

The three replacement policies that we implemented were:

1. FIFO
2. LSF_MAXS (Largest String First)
3. Random

Random replacement policies that we implemented uses a map which associates an index with the URL (named `IndexMap`). The index is 0 based. As mentioned previously, the map is internally implemented as a self-balancing binary tree and has a time complexity of O(log n) for searching, insertion and deletion. LSF makes use of a priority queue. The details of it are explained in the following few paragraphs. Each of our cache replacement policy replaces multiple elements if there is not enough free space to fit in the new page (provided that the size of the page is smaller than that of the cache).

1. FIFO (First In First Out) Replacement:

This method replaces the oldest value from the cache i.e. the first page in the cache is the first to be removed. We use the stl::map's `begin()` function to find the first element in the map. This cache should be performing well if we access recent pages. For example: if I access pages A then B then C and finally D (A → B → C → D) and if my access pattern is D → C or D → C → B → C → B or C → D → C, etc then my cache will perform well as it wouldn't have to fetch new pages. However, it will replace the oldest page if you access a new page. For ex: if you access a page E after D and suppose that the cache is at its limit then as per this cache replacement policy the page A is removed from the cache since it is the oldest entry in the cache. Suppose that E is a large page and just removing A doesn't create free space for E then B is removed from the cache to make space for E. This goes on till there is enough free space in the cache to fit E. However, these steps happen if and only if the page is small enough to fit in the cache. We can clearly see that accessing the recently visited pages would give us a good performance while accessing old pages would result in a performance loss. This is verified by our readings.

2. LSF (Largest String First) Replacement:

This method replaces the page which acquires most space from the cache. This makes sense because clearing up the page taking up the most space creates space for many small pages. We make use of a priority queue to keep track of the largest page. A node in the priority queue consists of the URL and the page size. The priority queue is sorted in descending order of page size i.e. the page with the largest size is present at the front of the queue. Using the URL, in the priority queue's node, we can find the particular element in our cache (the map). After finding the element we remove the entry from the cache as well as the priority queue. For example: if A → B → C → D are the pages accessed in order and say that their sizes are C > A > D > B, i.e. the priority queue arranges them in the order C → A → D → B, then if we want to make space for a new page, say E, in the cache, this replacement policy selects C to be replaced first for the page E. If, even after that, there is no space in the cache then

the A is replaced. This goes on till there is enough space for E to fit in the cache. The above steps take place only if E is small enough to fit in the cache.

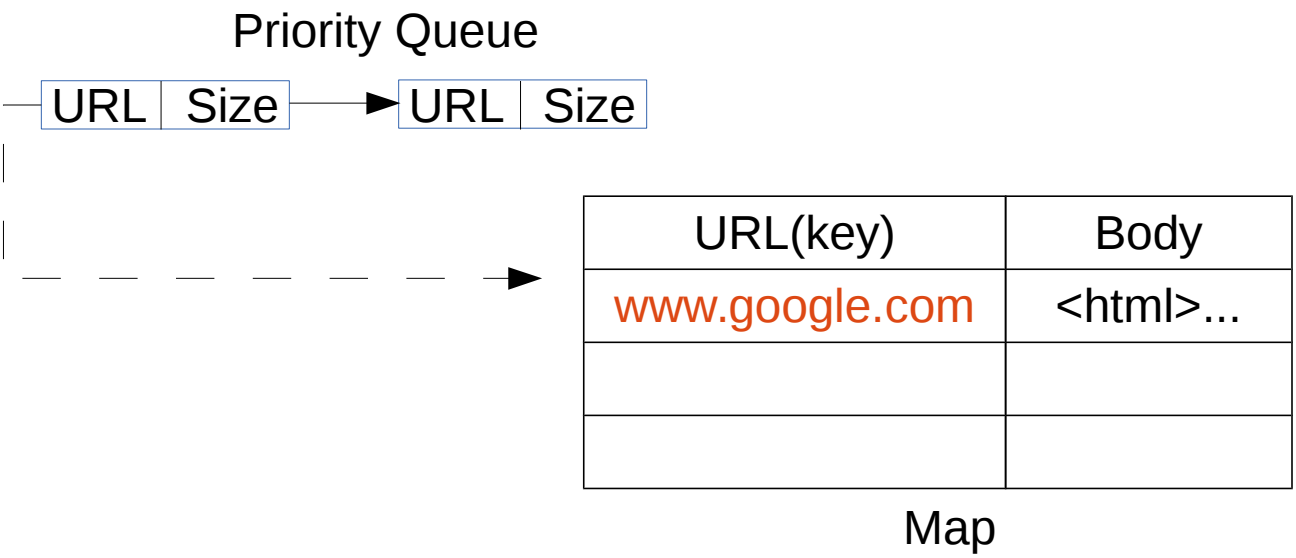


Fig: Priority Queue Sorted according to the page size and using the URL to find the entry into the map

3. Random Replacement:

This method selects a random index from the index map and removes the corresponding URL entry in the `IndexMap` as well as the `MyMap`. We use the `rand()` method to generate a random number and the random seed passed to that function is the current time (using `srand(NULL)`). Since this method selects a random page to replace from the cache, one cannot say anything about its performance. It should not necessarily perform very good but should not perform very bad either (this indeed happens to be the case as we see from the results). If the page doesn't fit in the cache then multiple elements are removed from the cache provided that the page itself fits in the cache.

Work Division

Detailed work division is as follows:

Task	**Member**
Understanding and implementing a sample client server program using Apache thrift	Both
Random Cache replacement policy	Both
FIFO cache replacement policy	Swapnil
MAXS cache replacement policy	Girish
Experiment Analysis (Collecting data)	Both
Documentation	Both

Evaluation

We implemented 3 cache replacement policy: Random algorithm, FIFO algorithm and Largest Size First (MAXS) algorithm. We conducted our experiments on machines running on platform Ubuntu 12.04 AMD64. For dataset of URLs, we selected a set of 70 unique URLs from sites such as wordpress.com and 100bestwebsites.org. Using this set of URLs, we generated 4 workloads for conducting experiments for the project. Each workload was created in such a way that it favored one cache replacement policy. In addition to this, we conducted our experiment on random set of URLs which were uniformly distributed. Experimental data is available in the 'Experiment Data' folder.

Workloads:

Workload	**Number of URLs (Dataset size)**
Workload 1 – Random set	500
Workload 2 – Favoring FIFO policy	600
Workload 3 – Favoring MAXS policy	188
Workload 4 – Random contiguous set (Uniform Random)	500

Detailed description of each workload and hypothesis about the policies is as follows:

1. **Workload 1**: We created a random set of 500 URLs from the dataset of 70 unique URLs. This workload was created to test the policies under random dataset which is expected in everyday traffic. We expected every policy to perform equally in this dataset.

2. **Workload 2**: This workload was created to test the effectiveness of FIFO policy. The workload favored the FIFO policy as it exhibited locality property where a URL was repeated with high frequency in a subset of dataset.

3. **Workload 3**: This workload was created to test the effectiveness of MAXS policy. This workload favored the MAXS policy. For creating this workload, we measured the size of the body returned on the http request of each of the URLs. We had chosen a unique set of 35 URLs for this purpose. Out of these URLs, there were URLs such as artforum.com, cnn.com etc whose page size were 10 times larger than other pages. We created a set of 188 URLs in which we inserted these high size URLs which were located at a large distance from each other in the dataset assuming the fact that users are less likely to re-access large documents because of the high access delay associated with such documents.

4. **Workload 4**: We tested our policies on uniformly distributed random workload. This workload consisted of 50 random URLs from the list of unique URLs. These 50 URLs were repeated 10 times to get a dataset of 500 URLs.

Experiment Metrics

We considered cache hit ratio and service request time (Latency) as performance metrics. Cache hit ratio was calculated as the ratio of number of cache hits to the total number of URLs in the dataset. Latency (Service time) was measured as the time taken for the requested body to reach the client after the URL was requested by the client. We measured the performance of our cache replacement policies using the above metrics by varying the cache size from 1MB to 5MB. Cache hit ratio is an important metric as it directly affects the performance of the cache replacement policy. Greater cache hit ratio means that relevant pages are in the cache and replacement policy is successfully removing those pages which are less likely to be requested. Similarly, the service time determines the time for the client request to complete. Greater the average service time means that the cache replacement policy is not doing its job correctly as there are frequent http requests to the original server which is time consuming process as it has to traverse the entire network. Also, we considered cache size as important metric because as the cache size increases, the hit ratio increases for a given policy because with the increase size cache can accommodate more pages which would result in more cache hits and thus better performance.

Experiment Description

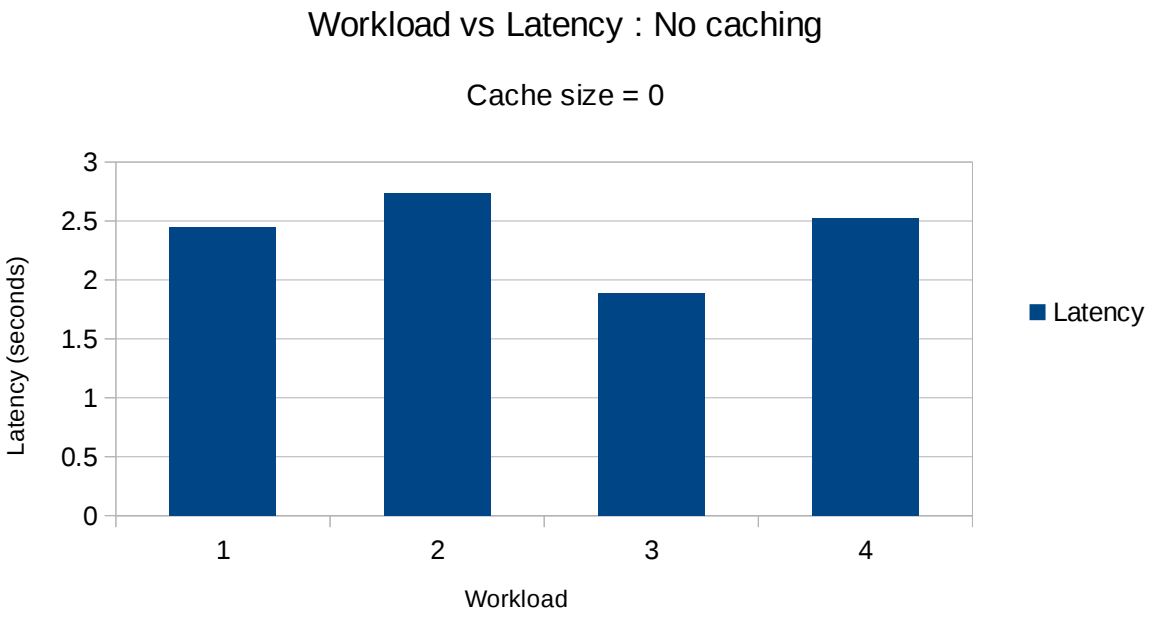
Experiments were performed on two machines running Ubuntu 12.04 AMD64. One acted as a server and other as a client. We connected the client and server using Ethernet with speed of 790MBPS. We measured cache hit ratio and service time metrics for evaluating the performance of our cache replacement policies. For measuring cache hit ratio, we maintained a static counter which incremented for every cache hit. So, for a dataset of URLs, we got number of cache hits which was divided with the number of URLs to get cache hit ratio. For determining service time, we called we started a timer before calling the server function and stopped the timer when the body of web page was returned. The time difference gave us the time spent in retrieving the body of the web page.

Experimental Results

All the raw data is available in 'Experiment Results' folder.

Experiment 1: No caching

The cache size was set to zero (no caching) which forced the proxy server to retrieve the body of the requested URL from the original server for each URL. We measured the average time taken for the proxy to return the body of the requested URL across four workloads.

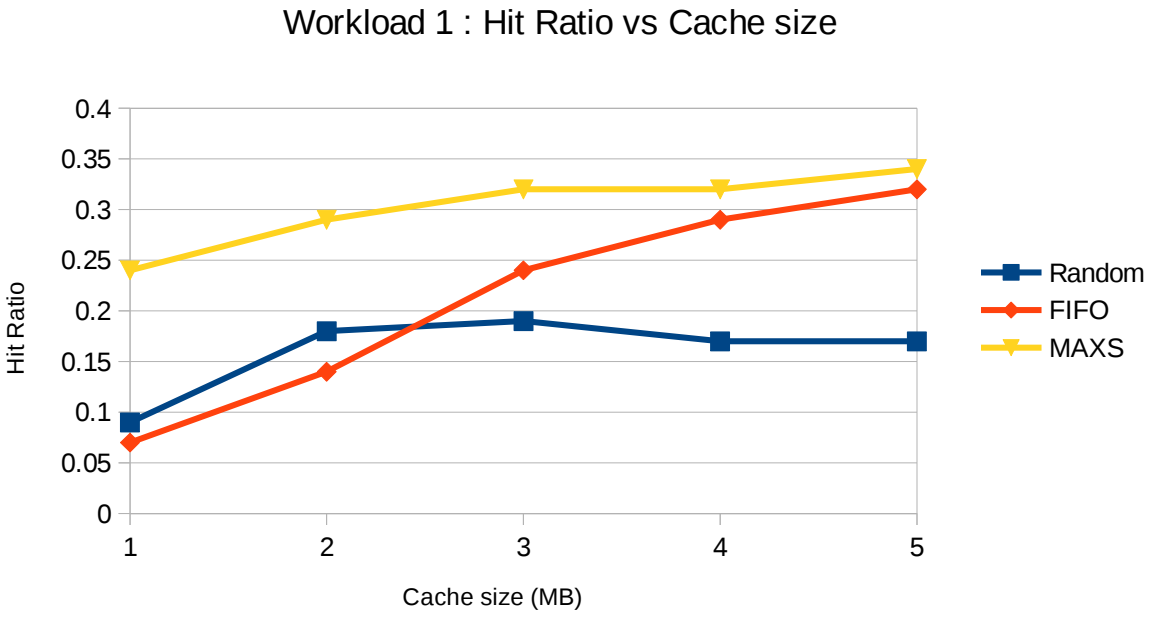


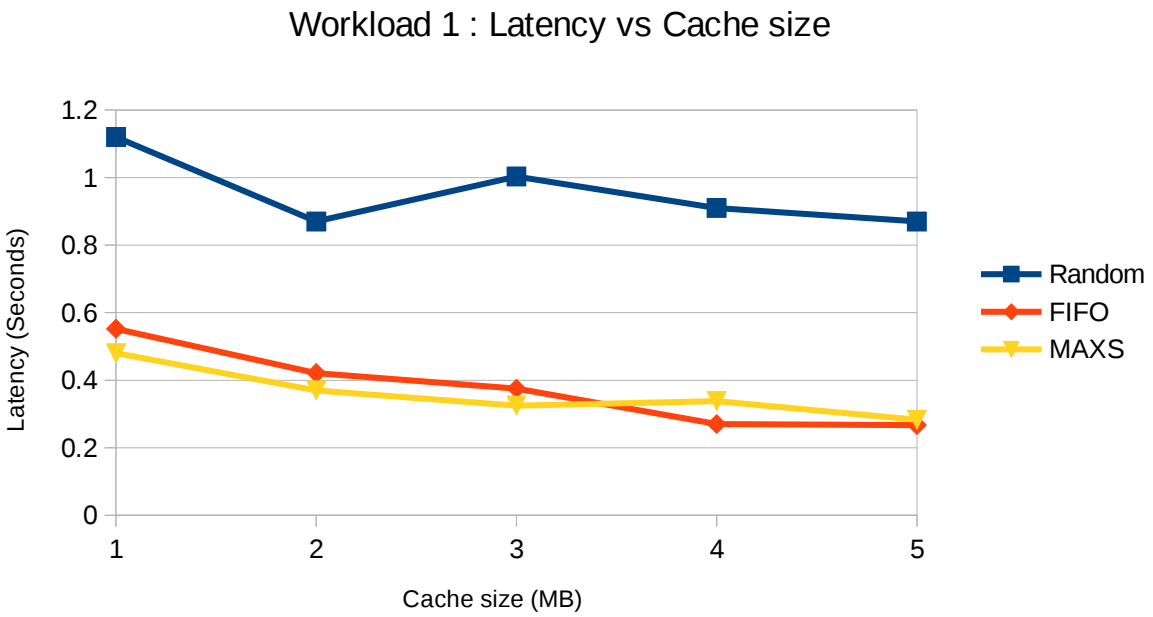
Inference:

1. As each workload contained different number of URLs and also the size of each URL's body was different, we got variable service time for each of the workloads.

Experiment 2: Workload 1 – Random set of URLs

The experiment was performed with random set of 500 URLs. The cache size was varied from 1MB to 5MB for each of the 3 policies. Hit ratio and service time (latency) was measured for each of the policies and a graph was plotted for hit ratio vs cache size and latency vs cache size.



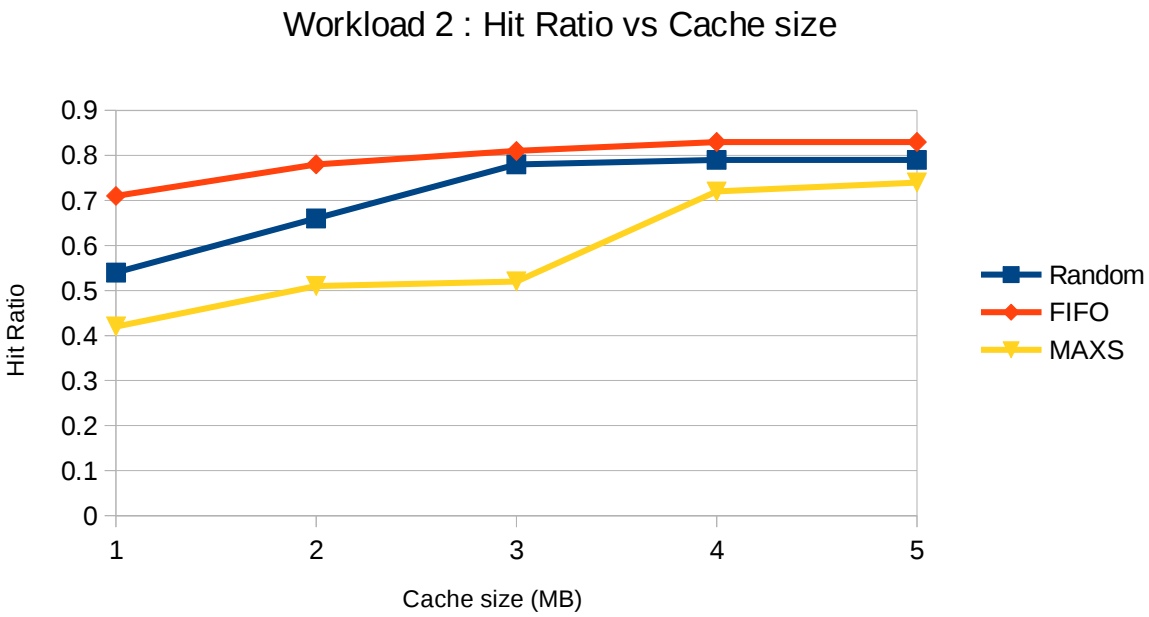


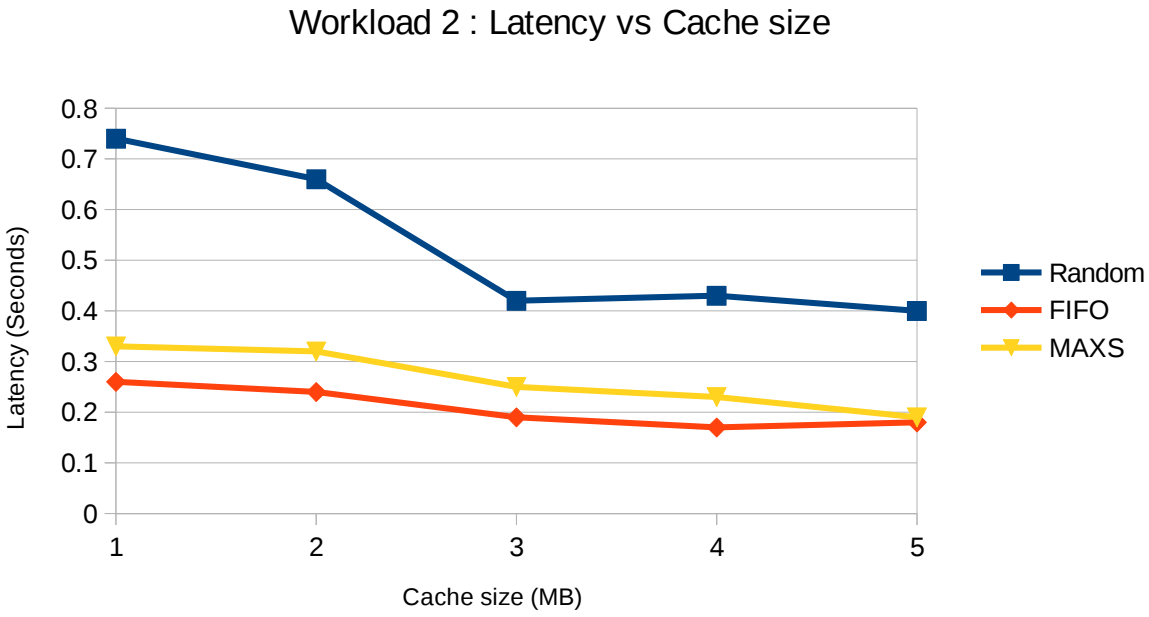
Inference:

1. In completely randomized workload, we see that all the three policies perform well in terms of hit ratio when the cache size is small.
2. As the cache size is increased, the hit rate for FIFO and MAXS increased as compared the Random whose hit rate , more or less, remained constant.
3. The reason for increase in hit ratio can be as the size of the cache increased, number of pages that cache can hold increased and the likelihood of the page already in the cache is increased.
4. In terms of latency, FIFO and MAXS outperformed random policy.

Experiment 3: Workload 2 - Favoring FIFO policy

The experiment was performed with workload of 600 URLs. The cache size was varied from 1MB to 5MB for each of the 3 policies. Hit ratio and service time (latency) was measured for each of the policies and a graph was plotted for hit ratio vs cache size and latency vs cache size.



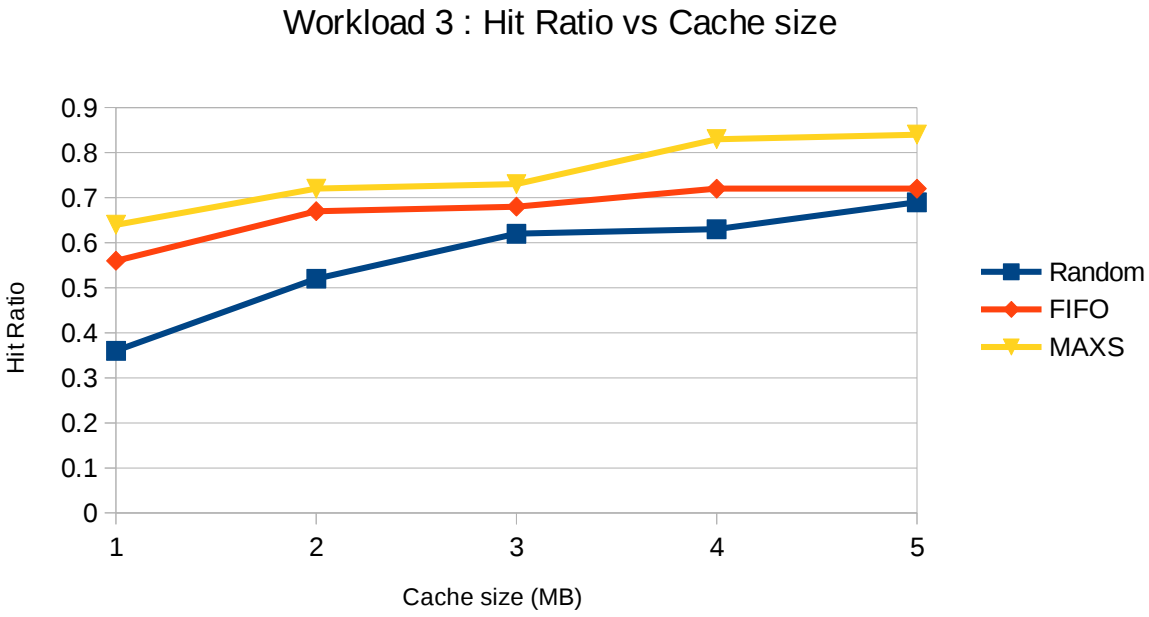


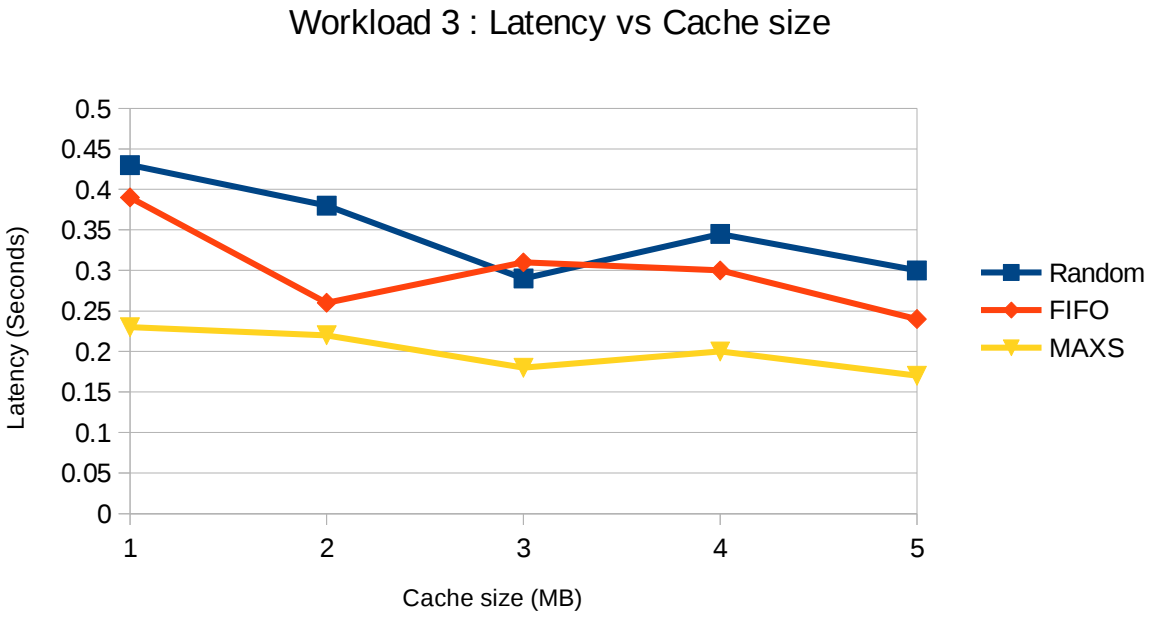
Inference:

1. As expected with the pattern of the locality based workload, the hit ratio is higher for FIFO policy as compared to random and MAXS policy.
2. But it can also be seen that as the cache size is increased the difference between the hit ratios of the policies decreased as the cache size approached 5MB.
3. Also, when FIFO policy was used, the service time was minimum followed by the service time of MAXS policy and then random policy.

Experiment 4: Workload 3 - Favoring MAXS policy

The experiment was performed with workload of 188 URLs. The cache size was varied from 1MB to 5MB for each of the 3 policies. Hit ratio and service time (latency) was measured for each of the policies and a graph was plotted for hit ratio vs cache size and latency vs cache size.



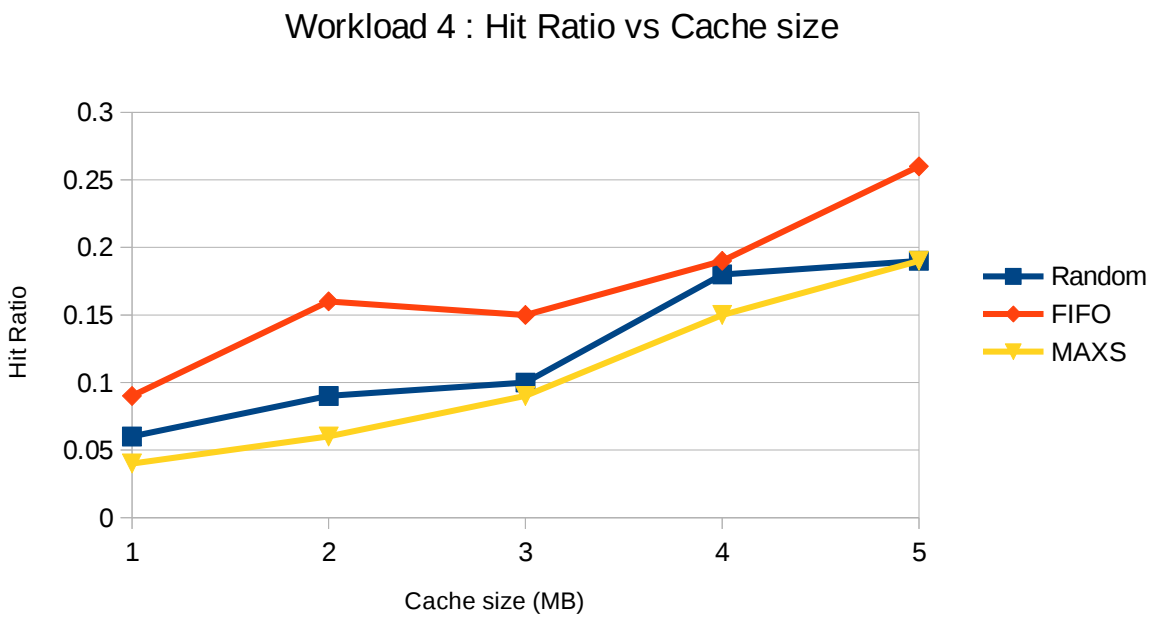


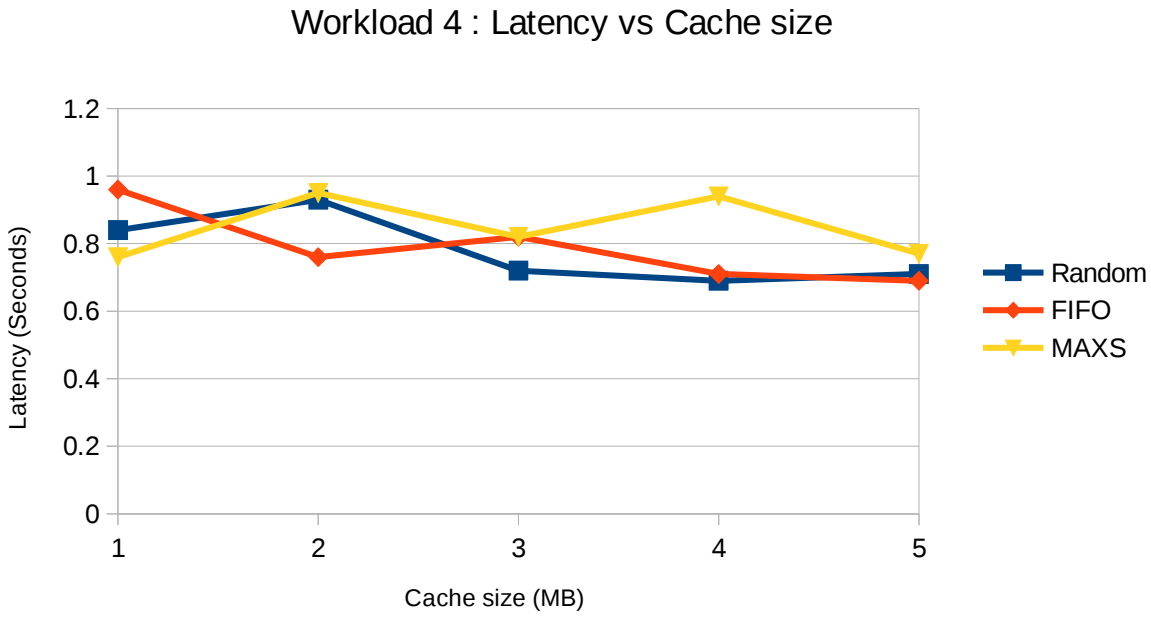
Inference:

1. As expected with the pattern of the size based workload, the hit ratio is higher for MAXS policy as compared to random and MAXS policy.
2. As seen with workload 2, as the cache size is increased the difference between the hit ratios of the policies decreased as the cache size approached 5MB.
3. Also, MAXS policy gave minimum service time followed by the service time of FIFO policy and then random policy.

Experiment 5: Workload 4 - Random contiguous set (Uniform Random)

The experiment was performed with workload of 500 URLs. The cache size was varied from 1MB to 5MB for each of the 3 policies. Hit ratio and service time (latency) was measured for each of the policies and a graph was plotted for hit ratio vs cache size and latency vs cache size.





Inference:

1. The frequently repeating patterns in this workload type favors all the three policies as clearly visible from the charts.
2. All the policies performed equally in terms of number of cache hits. With increase in cache size, the hit ratio was increased proportionally as expected.
3. The latency also showed similar trends where all the three policies performed equally across different cache sizes.

Conclusion:

Most of the web sites accessed by a user in real world are of small size (around 20KB). FIFO replacement policy is easy to implement and results in decent performance. FIFO works the best when the user visits same websites which he/she has visited recently. There might be cases when a user visits a website of large size. MAXS works the best when the dataset contains some large size pages which are rarely accessed by the user again due to its size. But MAXS leads to the worst performance when the majority of the URLs are of same size.