RPC-based Proxy Server

Team members

1. Aditya Kadur
2. Deepti Kochar

Introduction

In this project, we have designed a proxy server which communicates with clients over a network through remote procedure calls. The server accepts requests from clients, which contain URLs for webpages on the internet and supplies the corresponding HTML document for this URL. On the server end, we have used the **libcurl** library for fetching the HTML documents for a given URL while the client-server communication is handled via RPCs using **Apache thrift**. The server also implements a software cache for storing a results from previous requests. In cases when a request is found in the cache, the server counts it as a hit and supplies the data from the cache instead of fetching the HTML data from the internet. This is expected to provide significant performance improvement as communication over the internet is expected to be much slower than the proxy server-client communication. We have implemented three different caching policies on the server– Random, FIFO and LRU. We have also implemented a test client which sends requests to the server over a network in 2 different access patterns (random and staggered repetition) from a large input URL text file. We have performed a detailed performance evaluation of our system which tries to measure and compare the performance of our proxy server under varying cache sizes for different caching policies. We have also compared them against the no-cache (a cache size of 0) case.

This document provides a detailed description of our testing methodology, hypotheses, observations, inferences and explanation in the following sections. Our measurement metrics include hit rate, average hit time, average miss time, miss penalty and average access time. These metrics have been chosen based on standard research practices for hardware caches in computer caches. However, since this experiment deals with documents of varying size, we have also computed the more meaningful per-byte version of the same metrics.
We find that the LRU cache performs the best in most cases, with the details and reasoning discussed in the second-last section.

Caching Policies

We implemented the following types of caching:

1. Random Cache - This type of policy randomly selects an item to remove from the cache. There is no consideration for when the item was added, when it was used, the size etc. One advantage of this policy is that it has a better worst case performance under different loads. The main disadvantage is that it does not take advantage of locality.

2. FIFO Cache - This type of cache follows the “First In First Out” policy. It will always remove the oldest item which was added to it. The main benefit is that it is relatively easy to implement. We don’t need to search through the entries to decide which to replace if we maintain a queue. We can simply pop out the item at the front of the queue. The primary disadvantage is that three is a sharp cutoff for how much locality we can exploit as an element must be kicked out after a certain time (dependant on size of cache). Also, for large cache sizes, the efficiency decreases as it requires more overhead to maintain the queue.

3. LRU Cache - This caching policy removes the entry which was least recently used. Out of all the items in the cache, it selects the one which was last accessed before all the other items. The main benefit is that it takes advantage of locality. Although the LRU replacement policy is relatively efficient, it does require extra data to be stored to track when each block is accessed, and is relatively complex. Another problem with the LRU policy is that each time a cache hit or miss occurs the comparison and LRU queue shift operations require time and computation. As the number of items in the cache increases the overhead increases.

Cache Design and Description

Each cache is implemented as a class which contains the following.

Data members

1. size - Amount of cache space occupied currently
2. MAX_SIZE - Total size of the cache

3. dictionary - A map containing the mapping of urls to data. The dictionary is an std::map type which is structured like a tree and has O(logn) insert, find and erase time. This data member is present in the random and fifo cache.

Public Methods

1. cache_fetch - This takes a url as an input and a string passed by reference to get the data from the cache. It returns 0 if the item is not in the cache. If it is present, it returns 1 and sets the string which was passed by reference to the value of the data present in the cache. It simply uses the url as a key to access the map to get the data. This takes O(logn) time.
2. cache_insert - This takes as input 2 strings - url and data. It checks if the size of the data is greater than the total size of the cache. If yes, it return 0. Otherwise it will insert the item in the cache and return 1. To insert it, it runs a while loop which checks if the item will fit in the cache. If not, it decides which item to replace based on the replacement policy of this particular cache and removes that item. These steps take place till there is enough space in the cache to insert the new item.
3. cache_set_max - This is used to set the total size of the cache. It is called by the server with an integer as input.

There are also some private methods to find if an item exists in the cache, to check if the cache is full, to decide which item to replace, to remove an item etc.

1. Random Cache

It has additional data members:
1. num_entries – This keeps track of the number of entries in the cache.
2. MAX_NUM_ENTRIES – This is the maximum number of entries in the cache. We assume that the minimum entry size is around 10 KB so the maximum number of entries would be the total size of the cache divided by the minimum file size.
3. urls array – This is a dynamic string array of size equal to the maximum number of entries.

Replacement algorithm

The random cache stores an additional array which contains the urls of data stored in the cache. This helps to decide which url to replace in O(1) time. The replacement algorithm generates a random number which is less than the number of entries. It uses the urls array to find the url corresponding to this randomly generated index number.

This url is used as a key in the map which stores urls vs data. It retrieves the size of the data and decrements the currently occupied size of the cache by that amount, erases the entry from the map, replaces the url at the randomly generated location by the last url in the array and then decrements the number of entries. Accessing the map to find the data and erase it is O(logn) while the other operations are O(1). So the decide and remove time for an entry is O(logn).

Pseudo code:

```
index = random() % num_entries
url_to_remove = urls[index]
size_data = dictionary[url_to_remove].length
size = size – size_data
dictionary.erase(url_to_remove)
urls[index] = urls[num_entries-1]
num_entries = num_entries – 1
```

Insertion

To insert an entry into the cache, this cache follows the algorithm described above (for cache_insert) i.e remove items till there is enough space for the new item. When there is enough space in the cache, it does the following steps to add the new item to the cache:

```
dictionary[url] = data
size = size + data_size
urls[num_entries] = url
num_entries++
```

The above 4 steps take O(logn) time since the map access takes O(logn) time.

Fetch

The fetch operation is the same as the one described in cache_fetch.

2.FIFO Cache

It has one additional data structure – a queue (called fifo_queue) to maintain the First In First Out principle. We used an std::queue structure storing the string urls. It has O(1) push and pop operations. (This cache does not need to store the number of entries

unlike the random cache.) We keep the map which stores the url vs data mappings to keep the fetch time O(logn).

Replacement Algorithm

Deciding which item to replace is again very straightforward for this cache. We simply select the one at the front of the queue.

Pseudocode:

```
url_to_remove = fifo_queue.front()
data_size = dictionary[url_to_remove].length
size = size – data_size
dictionary.erase(url_to_remove)
fifo_queue.pop()
```

The queue operations are O(1) while the map access is O(logn). Therefore the time complexity is O(logn).

Insertion

To insert an item in the queue, we run the replacement algorithm till there is enough space for the new item. Then we add this item to the map and push it into the queue.

```
dictionary[url] = value;
size = size + data_size;
fifo_queue.push(url);
```

Fetch

The fetch operation is the same as the one described in cache_fetch.

3. LRU Cache

It has additional data structures –
1. A double-linked list queue (called lru_queue) to maintain the Least Recently Used principle. We used our own queue data structure (defined in lru_queue.h) for storing the urls and data. It has O(1) push and pop operations.
2. A map (called queue_ptrs) of the urls vs the pointers to the queue elements. This is used to make the access time to any item O(logn) – map access time is O(logn) and

accessing the element pointed to by the corresponding pointer is O(1). This is useful for the fetch operation.

Replacement Algorithm

Deciding which item to replace is very straightforward for this cache. We simply select the one at the front of the queue.

Pseudocode:

```
url_to_remove = lru_queue.front_url()
data_size = lru_queue.front_size()
size = size – data_size
queue_ptrs.erase(url_to_remove)
fifo_queue.pop()
```

The queue operations are O(1) while the map access is O(logn). Therefore the operation is O(logn) time.

Insertion

To insert an item in the queue, we run the replacement algorithm till there is enough space for the new item. Then we add this item to the map and push it into the queue.

```
size = size + data_size;
queue_element *ptr = new queue_element();
ptr->url = url;
ptr->data = value;
lru_queue.push(ptr);
queue_ptrs[url] = ptr;
```

Fetch

The fetch operation is the slightly different from the other fetch operations. The algorithm looks through the queue pointers map using the url as a key to get the pointer to the queue element. It then removes the element from this position by changing the previous and next pointers of the adjacent elements and pushes this element at the back of the queue (all this is done by the lru_queue function send_to_end). This is done to maintain the Least Recently Used policy – the items which were least recently used are at the front of the queue while the ones which were most recently used are pushed to the back.

Evaluation Metrics

Based on similar research on hardware caches in the field of computer architecture, we employed the following metrics for our performance evaluation. However, one key difference here is that the data supplied for each request has widely varying sizes. Thus, we have used a per-byte version of the same metrics where applicable.

1. Hit rate – Ratio of number of requests which were cache hits to the total number of requests.
2. Data hit rate – Ratio of total numbers of bytes read from cache (hits) to the total number of bytes requested. The hit rate is accurate indicator of how effective a cache was in utilizing results of previous requests and maximizing utilization of cached data. The higher the hit rate, the more effective the cache, and the better the performance.
3. Miss penalty per kB – Average time taken for the server to supply a kilobyte of response data in case of a cache miss. This metric is used to give an indication of the cost associated with a cache miss. This metric is largely dependent on the network performance (internet bandwidth as well as the resource being accessed). However, if we assume that the average cost of accessing an internet-resource is constant, we can get an estimate of time taken to repair a cache miss, i.e., fetch the web page, store it in the cache by possibly evicting one entry, and provide the data to the client.
4. Average Access Time (AAT) per kB – The overall average time taken to transfer 1 kB of response data from server to client. This includes both hits as well as misses. This is the overall key performance metric for any cache as it gives an idea of the overall performance of the cache.

Workload Description

Our workloads were generated by the client from a URL list file. The URLs are read into a vector by the client which accesses them in a pattern as described below, for a specified total number of times (much larger than the number of URLs).

The URL list initially contained 250 URLs which were selected from Alexa's top websites list. We first tried limited runs using 1000 total accesses by the client. However, these were extremely time-consuming. Moreover, the result trends were not any different than that indicated by a lower number of accesses. Thus, we made the final set of experiments using a 50 URL file with 500 requests by the client in each run.

We have used the following 2 access patterns for our workloads:

1. Random access - An entry from the URL vector is selected at random. The probability of selection for all every URL is the same. This pattern is used to simulate a perfectly arbitrary sequence of accesses.

2. Staggered repetition – We chose 3 high-priority URLs from our vector and ensure that these have a regular frequency of being accessed, i.e, every third request is URL-1 , every fifth request is URL-2 and every seventh request is URL-3, with priority to higher numbered URL (e.g- the 15th access will be URL-2, not URL-1). The remaining accesses are chosen at random from the entire URL list. The 3 high-priority URLs themselves are chosen randomly, while ensuring mutual non-repetition. After every 50 requests, 3 URLs are chosen again at random. This pattern simulates high temporal locality. We have tried to add some randomness to the accesses while ensuring that there is still a constant high degree repetition of requests. An effective cahing policy should be able to service a high number of hits with this pattern.

Experiment Description

Hardware setup:

· We used our personal laptops, each running Ubuntu 14.04 64-bit for all steps of this project.

· One of the machines acted as the client, while the other acted as server.

· For our experimentation, we ran our tests on a home network with the client and server connected to each other via WiFi, which also provided the internet connection to the server. The aggregate bandwidth for the internet connection was 30mbps.

Software used:

· All code for both server and client was written in C++.

· For compiling our code, we used Apache Thrift v0.9.2 g++ v4.9.1 and Libcurl.

Experimentation

- For experimentation, we ran our tests with a varying number of input URLs and varying total number of accesses. However, we observed that the ideal balance between experiment length and appreciable trends was observed for URL list of 50 entries and total 500 requests per experiment.
- We varied the cache size between 0 and 16MB.
- We used a large number of metrics including counts of all hits, misses, cache overflows (document too large to fit in cache), failures (Curl failed to fetch HTML document), the size of these files and the time taken for each. Some of these metrics are plotted below, as explained in earlier sections.
- For timing measurements, we used the gettimeofday function and stored all cumulative times with microsecond precision.

We used the 2 different access patterns for each of the 3 different cache types - Random, LRU and FIFO to make a total of 6 test cases.
These 6 test cases were run for each of the different cache size - 0.5 MB, 1 MB, 2 MB, 4 MB, 8 MB and 16 MB.
We also ran the 2 different access patterns for a size 0 cache (no cache).

Expectations:

- Random access - We expected that for the random access case, the FIFO and LRU caches would perform similarly with increasing hit rate as the size of the cache increased. Consequently, the AAT would decrease while the miss penalty remained constant. More importantly, we anticipated a random cache would work slightly better since the others are designed to exploit a certain access pattern while random is well, random.
- Staggered repetition - We expect the LRU cache to outperform random cache by a notable measure in this case since the pattern exhibits high temporal locality of accesses and the LRU cache is designed to exploit the same. We also expect the FIFO cache to perform the worst here since although it can benefit from limited locality, it has a sharp cutoff on the limit to which it can accept locality before encountering a cache miss depending on the size of the cache. However, for larger cache sizes, we anticipated comparable performance of FIFO with LRU and random.
- We also expected the miss penalty for every cache to be more or less similar since insertion time is O(logN) for all of them.

Experimental Results

The following graphs present our experimental results. The raw data containing all our metrics can be found in the tarball (labelled Results.txt).

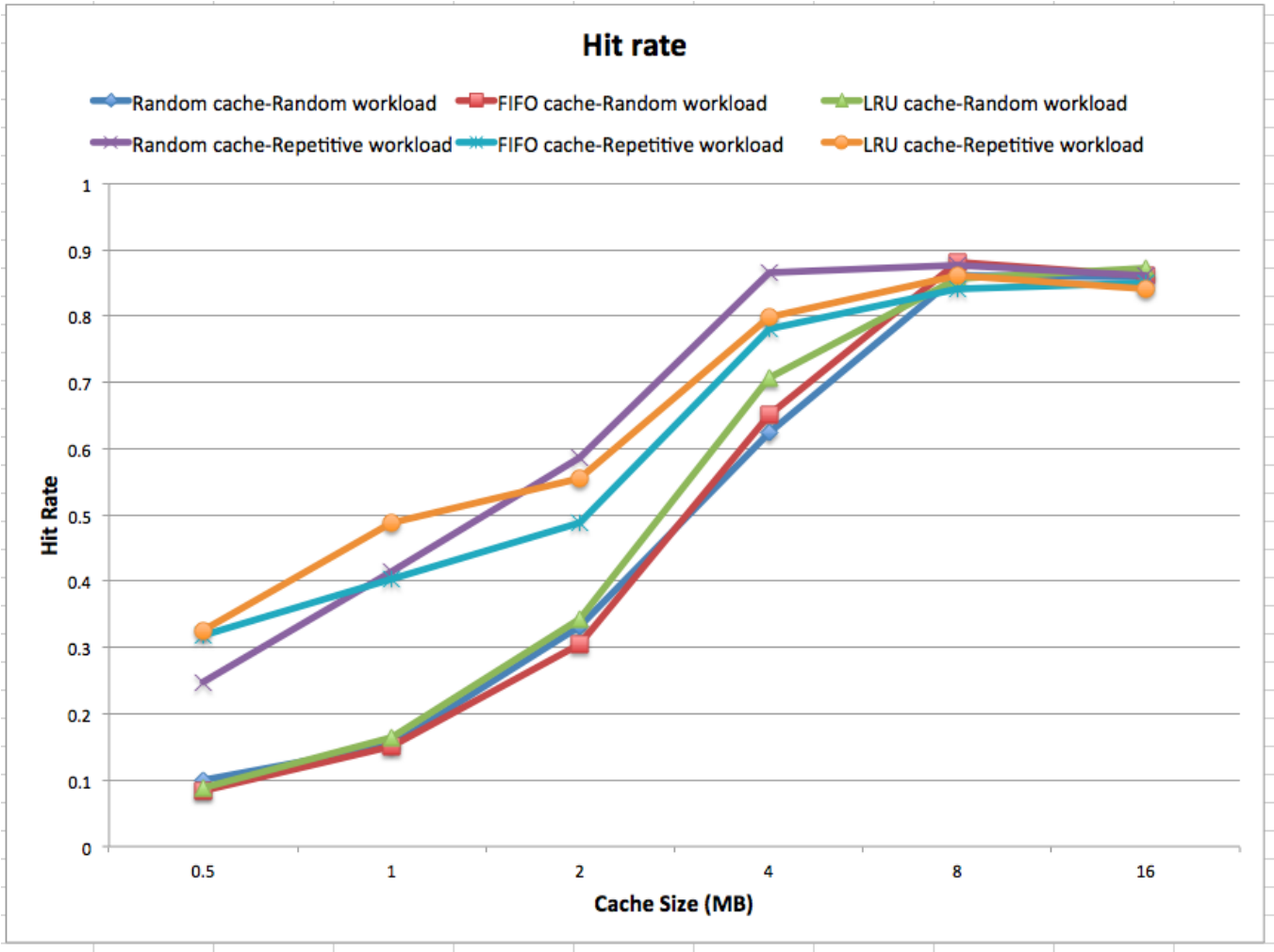

Figure 1. Hit rate: Number of hits/number of requests vs Cache Size

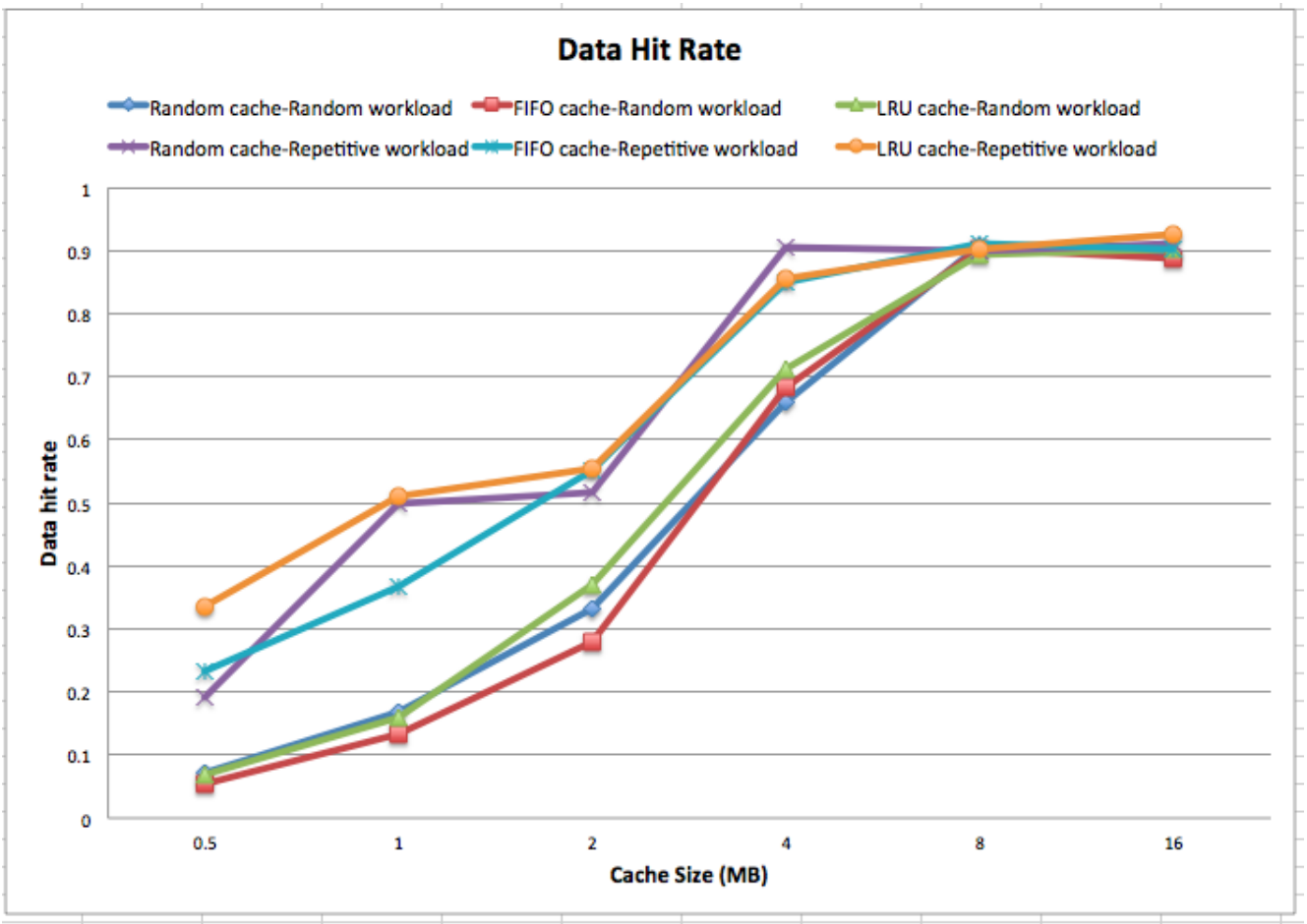

Figure 2. Data Hit rate: Number of hit bytes/number of bytes requested vs Cache Size

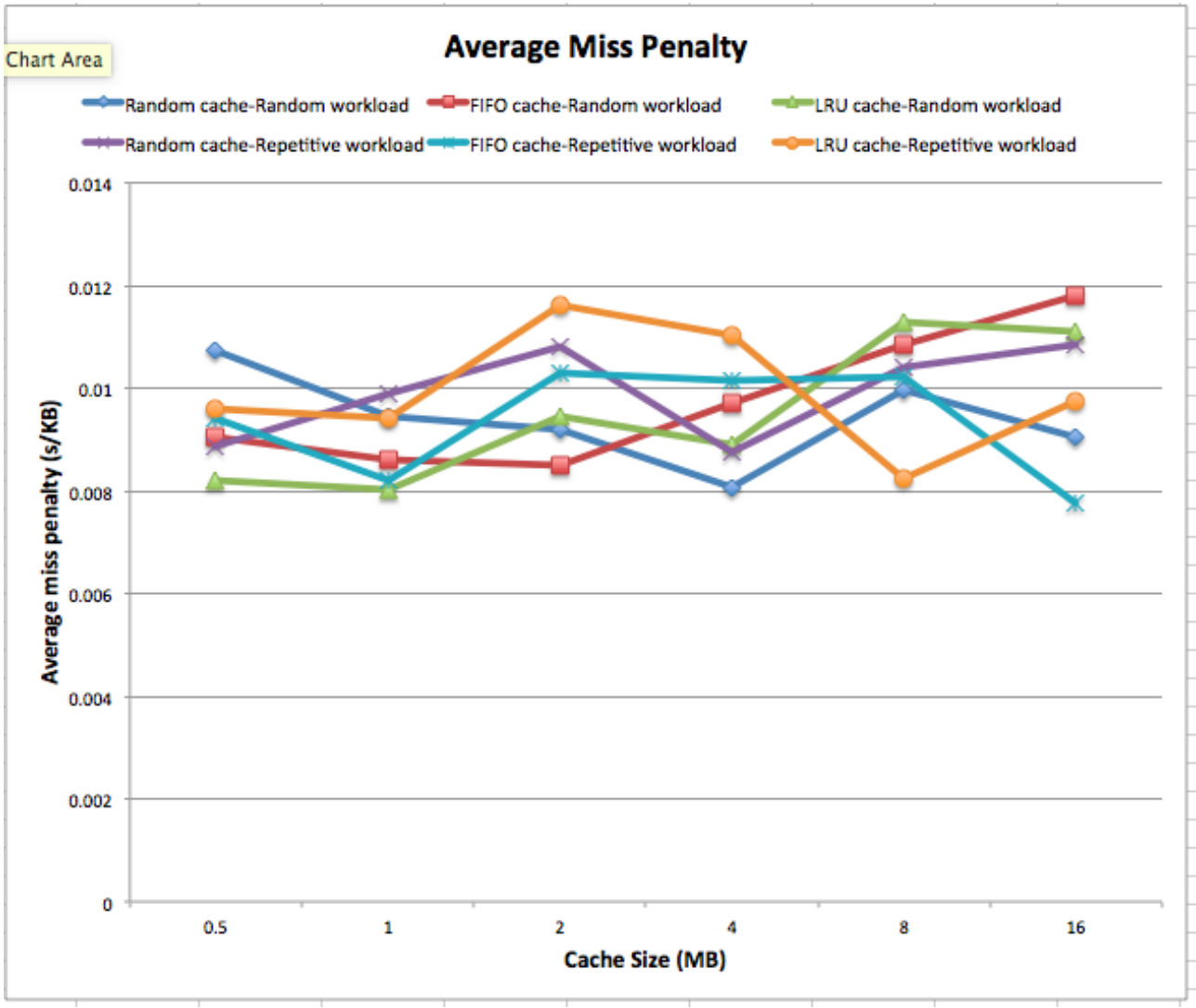

Figure 3. Average Miss Penalty: Number of seconds per KB vs Cache Size

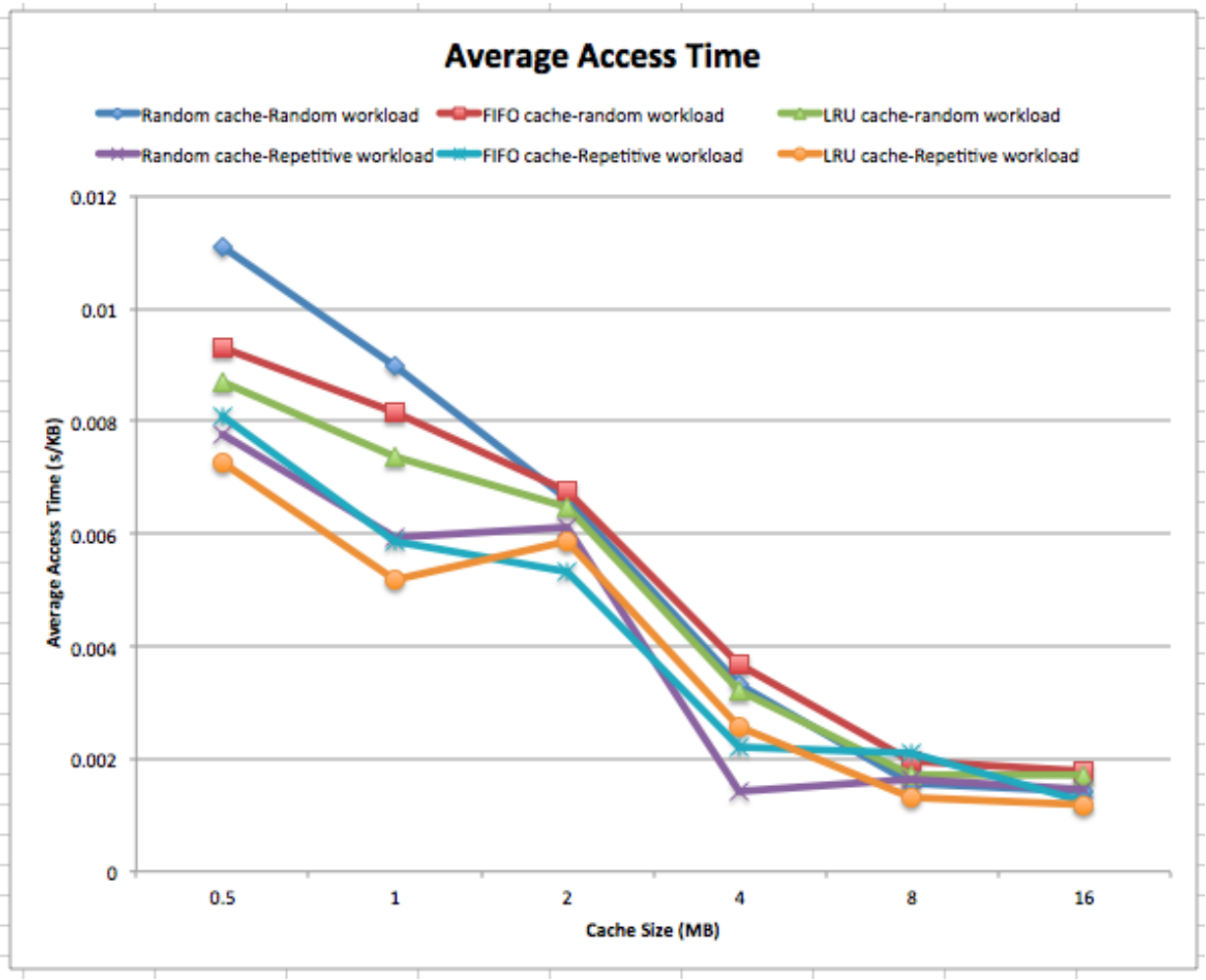

Figure 4. Average Access Time: Average access time in seconds per KB vs Cache size

Results with cache size of 0

Random - Miss penalty per kB = 0.00955147
Repetitive - Miss penalty per kB = 0.00903713
AAT is equal to the miss penalty.

Analysis of Results

From the hit-rate graph:
We observe that the caches perform much better overall for the staggered repetition case in comparison to the random case. This was expected since the former exhibits high temporal locality which the caches can exploit, while the later does not.
With random access, we expected random cache to perform slightly better. However, all the caches performed roughly the same. In the given graph LRU seems to perform slightly better but this was not consistent when we repeated the experiment. The

inference here is that a random access pattern does not give any particular opportunity for better cache performance to any type of cache, including random cache, which rely on some sort of locality of accesses for better performance.
With the repetitive pattern, the LRU cache performed best while the FIFO cache performed worst as expected. However, the difference in performance was not as large as expected. Also, the random cache even outperforms LRU for one cache size. This again was since the pattern exhibits high temporal locality of accesses and the LRU cache is designed to exploit the same. The FIFO cache to perform the worst here since it has a sharp cutoff on the limit to which it can accept locality before encountering a cache miss depending on the size of the cache.
As expected, as the cache size increases, the performance of all caches converges. This is because the effect of replacement-policy begins to be overshadowed by the effect of the sheer number of entries in the cache. The temporal locality is almost taken always exploited by the contents of the cache irrespective of the replacement policy.
For a 0 size cache, the rate is understandably 0.
From the data hit-rate graph:
This graph was plotted to portray a more realistic representation of the cache performance measured in terms of bytes.
As can be observed, the graph is very similar to the hit-rate graph and the observations and inferences remain the same.
From average miss penalty (per kB) graph:
We expected all caches to show similar results here. This is indeed the case as can be observed. This is because all caches have O(logN) insertion time in our implementation. Moreover, this metric is highly dependant on the network conditions and since, it extremely difficult to have constant network conditions over the internet for an extended time, it is highly unlikely we'd see any notable trends in this metric.
The value for 0 sized cache is also the same since this is largely a network dependant parameter.
From Average access time (per kB) graph:
Since the miss time dominates over hit time (metrics are part of the complete results in attached tarball), we expect this graph to follow the inverse of the hit-rate graph. This is indeed the case as access time in repetition access pattern are in general lesser than that of random access. The difference is more notable for smaller caches, with LRU performing the best.
This is primarily because the hit-rate for these caches is higher in these cases. The graph has an inverse correlation with the hit-rate graph.
For a 0 sized cache, the average access time is the same as the miss penalty since all the results are misses.

Conclusion

In this work, we have designed a RPC base proxy server in which a client interfaces with a server over a network and requests the HTML document for a given URL. We have also implemented 3 different cache replacement policies and evaluated the performance of the each experimentally.

We observe that the LRU cache offers significantly better performance amongst the 3 in case of accesses sequence with high repetition. Moreover, even in random traces, the cache doesn't perform notably worse or consistently roughly similar to other policies. We also note that the as the size of the cache increases, the hit rate increases.