CS-6210: Advanced Operating System

Project 3: RPC-Based Proxy Server Report

Saniya Ambardekar

Saumya Dwivedi

Introduction

In this project, we have built a web proxy server was built, to measure and compare performance of different caching policies. Current set of experiments focus on maximizing the performance of a proxy cache that has a limited amount of cache space, and thus experiences a lot of contention. We have implemented five caching mechanisms Random, LRU, LFU, LIFO and FIFO compared their performance for varying types of workloads: uniformly distributed and random. Cache Hit Rate, Latency and total number of requests are used to measure performance of these caching algorithms. We find that these algorithms perform variably depending on the request size and latency parameters but typically depends on the testing workload.

Cache Design Description

The web proxy cache mainly consists of server and clients which communicate via Remote Procedure call based on Apache Thrift. Server handles the caching mechanism configurable through the Makefile. Cache size and capacity can be handled by constants in the gtcache header file.

Clients submit the request to the server via a workload file - this consists of the URL and the number of bytes of data requested from the server. Functions to calculate the latency and hit rate are embedded in the server and web data response is returned to the client. Each response is stored in the file 'out.txt' in the client filesystem.

All the operations of the cache are declared in header file ('gtcache.h'). They are defined as:

1. Gtcache_init(): Called from the constructor of the server function. The gtcache_init procedure initializes data structures needed for the cache. In our case, that defines and initializes the cache vector, hash map, stack and the priority queue. The parameter capacity supplied to it controls the capacity of the cache which initializes the vector size. Metadata of all the cache entries is also initialized.
2. Gtcache_set(): This function is use to set or update the cache entries. In case of a cache miss, proxy server makes a call to this function and associates the data retrieved by curl operation to the given URL key. If the data exceeds the cache limit or there are no available cache entry ids an appropriate error is raised. In case of an update, the priority queue is also updated.
3. Gtcache_get(): This function retrieves the data associated with a given url. In case the data is not present in the cache, it returns NULL to the server, meaning it is a cache miss. If the parameter val_size is NULL, it is ignored. Otherwise, the length of the data array is stored in it. Eviction queue and number of hits are updated in the hash table.
4. Gtcache_memused(): The gtcache_memused procedure returns the sum of the sizes of the entries in the cache. There is a check to ensure that this amount should never exceed capacity.

5. Gtcache_destroy(): Function called when the server exits. It clears the eviction queue, hash table, memory allocated to metadata of cache entries and the stack.

Depending upon different cache replacement policies, the VAR variable in the description below will vary. For example, in case of LFU- total number of hits, LRU- time of entry in the cache etc. Implementing the LFU caching policy first, made it significantly easier to implement the rest of the policies.

Our cache replacement library uses ***multiple data structures***:

1. Data struct cache_entry_t: to hold the cache entry metadata i.e. Size, VAR, actual cache content and unique id to identify the entry.
2. Vector: A Vector of structs <cache_entry_t> to store actual cache entries. Size of this cache is given as an argument to the server.
3. Pair: A pair of VAR and unique cache entry id is used to uniquely identify the cache entries in the priority queue and make the the cache update consistent across all data structures and sub-linear in complexity.
4. Un-Ordered Map: The Hash table which stores the mapping between the url and pair<VAR + cache entry id>. This allows for constant time lookup [O(1)] of the url to cache entry id mapping in case of a cache hit.
5. Ordered Map: Ordered Map in C++ keeps all its entries ordered by the indexing key, in this case it is the pair<VAR + cache entry id>. In case of equal values of VAR, an entry with lower cache entry id is evicted. Ordered Map acts like a priority queue in LRU, LFU, FIFO and gives a worst time lookup of O(log N) where N is the size of the queue.
6. Stack: A stack is used to store the current available list of cache entry ids. As the cache entries continuously evict and new entries take their place, it is essential to have a data structure which give O(1) access to the next available slot.

GT Caching Server Miscellaneous properties

1. WWW_MIN_CACHE_LEN defines minimum cache length (default 1024) and CACHE_SIZE defines the number of entries (varied between 8-64). Hence total capacity of cache remains WWW_MIN_CACHE_LEN * CACHE_SIZE(default 1024*8)
2. Error Handling :
    a. gtcache_set: unable to create cache_entry : This happens when the the total number of cache entries have exceeded their limit (CACHE_SIZE = capacity/min_cache_len) even where there is some room in the cache. This is avoided by adding an extra check if there are available ids in addition to space in the cache.
    b. Not found in priority queue : If an entry found in hash map is not found in the priority queue, it is an inconsistent state for the server and error state is returned.
    c. Value exceeds cache limit: If a potential cache entry takes space insufficient for the cache, this error state is returned.

3. If a client requests for a specific amount of data, the parameter val_size is used in gtcache_set and gtcache_get. A val size of '0' indicates that the user wants the entire webpage, whereas a specific number of bytes are returned otherwise. If the entry exists in the cache, but the data size is not enough for the request, it is considered as a miss and a fresh request is made via curl and set in the cache.

Caching Policies Description

LFU:

Least Frequently Used caching policy keeps track of the number of times a block is referenced in memory. When the cache is full and requires more room, the system will purge the item with the lowest reference frequency.

One of the simplest implementation of LFU algorithm uses a counter for every block that is loaded into the cache. Each time a reference is made to that block the counter is increased by one. When the cache reaches capacity and has a new block waiting to be inserted the algorithm evicts the block with the lowest number of hits. LFU cache implementation requires three data structures. One is a hash table which is used to cache the key/values so that given a key we can retrieve the cache entry at O(1). Second one is a priority queue for each frequency of access to control eviction, i.e on evicting a cache entry, a new entry can be easily prioritized in time O(log N). The third data structure is a stack which controls the list of available ids. Insertion is done in O(1) time. Any new entry is added to hash table as well as to the priority queue. This is done in O(1). If the cache reaches its maximum size, existing entries are evicted in order of the frequency that they appear, from the lowest frequency and to the next frequency list and so on.

LFU suffers from the following two disadvantages:

1. Consider an item that is accessed frequently for a short period of time and then not accessed for an extended period. Such an item would have a high frequency due to frequent accesses early on, and would not be evicted even when it is not being accessed anymore. One sample load for such a scenario would be "A-A-A-A-A-B-C-B-B-C-D.." with cache size of 3. Here, although A is no longer being accessed, it will not get evicted due to the frequent accesses in the beginning. This would also result in the eviction of C, which might get accessed more frequently going forward.
2. New items that are added to the cache start with a lower counter value, and will get evicted very often, even if they are accessed frequently. In the implementation, the newly added entry is generally evicted first, owing to the last addition.

LRU:

Least Recently Used caching policy exploits the principle of temporal locality and keeps track of the time a block is referenced in memory. When the cache is full and requires more room, the system will purge the item that has not been used the longest LRU works on the

idea that pages that have been most heavily used in the past few instructions are most likely to be used heavily in the next few instructions too.
It uses a hash table to cache the entries and an ordered map to keep track of the access order. When the cache reaches its maximum size, the least recently used entry will be evicted from the cache using the ordered map. Whenever a url is accessed, it acquires the value equal to the counter at the time of url request. Whenever a url entry needs to be replaced, the application selects the entry with the lowest time counter and swaps it out for a new entry.
While LRU can provide near-optimal performance in theory, it is expensive to implement in practice. Its performance tends to degenerate under many quite common reference patterns. For example, if the LRU cache size is N, an application executing N + 1 subsequent requests will cause a cache miss on each and every access. For the access pattern "A-B-C-D-A-B-C-D-A-B-C-D.." with cache size=3, each memory access will result in a miss.

Random

Random replacement algorithm replaces a random cache entry in memory, in case eviction is needed to bring in a new url into cache.
Random cache replacement eliminates the overhead cost of tracking and updating URL frequency count or access time. It usually fares better than FIFO for its average evictions scenario, and for updating references it is better than LRU/LFU, although generally LRU performs better in practice. Random replacement gives better worst-case performance than LRU. A classic example where Random is better than LRU and FIFO is a repeated linear url access pattern slightly larger than the cache size (for example: "A-B-C-D-A-B-C-D.." with cache size = 3). In this case, both LRU and FIFO will be disadvantageous and end up dropping each entry just before it is needed. Random replacement on the other hand has a better chance of performing well on this access pattern.

FIFO

First-In First-Out Replacement algorithm is one of the simplest cache-replacement algorithms. It has a low-overhead algorithm that requires little bookkeeping on the part of the application. The application keeps track of all the entries in memory in a queue, with the most recent arrival at the back, and the oldest arrival in front. When a cache entry needs to be replaced, the entry at the front of the queue (the oldest entry) is selected. It does not require expensive data structures like priority queue to keep track of eviction scenario. While FIFO is cheap and intuitive, it performs poorly in practical application. Thus, it is rarely used in its unmodified form. This algorithm experiences Bélády's anomaly, where increasing the cache size results in an increase in the number of cache misses for certain memory access patterns (for example request pattern of d-c-b-a-d-c-e-d-c-b-a-e perform worse for cache size 16 than cache size 8 MB).

LIFO

Last-In First-Out cache algorithm keeps track of all the requests in memory in a map, with the most recent one at the top. It evicts the entry that is on the top and then push the new url that was brought in the cache on the map. It is also a low-overhead algorithm that requires little bookkeeping on the part of the application. It seems counter-intuitive, and is a good fit only for very predictable memory access patterns.

Metrics for Evaluation

The following two metrics were chosen to evaluate the performance of the implemented caches:

1. Cache Hit Rate: A cache hit occurs when the requested data can be found in a cache, while a cache miss occurs when it cannot. Cache hits are served by reading data from the cache, which is faster than recomputing a result or reading from a slower data store; thus, the more requests can be served from the cache, the faster the system performs. The hit rate is the percentage of all requests that can be satisfied by searching the cache for a copy of the requested object. Suppose a total of N calls are made to a server, and M of those calls are served from the cache. Then the corresponding hit rate will be M/N. Hit rate can be used to gauge the effectiveness of caching layer. Hit rate is affected by cache size, request patterns and page eviction policies. Some page eviction policies perform better with some access patterns. For instance, LFU performs better in case of caching of popular web-pages, with more or less consistent requests.
2. Latency: Latency is the delay from input into a system to desired outcome. In our case, it is the amount of time from requesting data for a URL to getting corresponding data. Latency is compared between a scenario which uses a web proxy cache, vs one which doesn't. Latency is hugely affected by the cache hit rate and the caching policy in turn. In general it is expected that a higher cache hit rate will imply a lower latency, but this also depends on the network conditions.

Cache Byte Hit Rate is another commonly used metric for web proxy cache performance. It is the percentage of all the bytes satisfied by searching the cache for a copy of the requested resource. Although it sounds similar to Cache Hit Rate, the Byte Hit Rate can vary greatly, as it also depends on the size of each cache entry. For instance, if the cache is populated by mostly small entries, it might have a higher cache hit rate even if some cache misses are seen for large objects. However, cache misses on large objects will adversely affect the cache byte hit rate.

Workloads Description

Some of the workload characteristics that impact proxy performance and cache replacement decisions are discussed below:

1. Cacheable Objects: It is important that the size of a requested object be less than total size of the cache, otherwise it is always considered as a cache miss.
2. Object Set Size: We used varying datasets varying in size (50 and 200) to determine the caching performance. The cache size is varied from 8 MB to 64 MB. It is important to measure cache performance with a high cache size to closely mimic practical scenarios. Due to the extremely large object set size the proxy cache must be able to quickly determine whether a requested object is cached to reduce response latency. The proxy must also efficiently update its state on a cache hit, miss or replacement.
3. Object Sizes: Although most of the requested objects are small, there are some extremely large objects available. The largest object requested during the measurement period was 35659 bytes. It is generally better to cache a number of smaller objects rather than a few large objects for better cache hit rate. However, this can be bad for the cache byte hit rate.
4. Recency of Reference: LRU works best when the access stream exhibits strong temporal locality or recency of reference (i.e., objects which have recently been referenced are likely to be re-referenced in the near future). Studies show that one-third of all re-references to an object occurred within one hour of the previous reference to the same object.
5. Frequency of Reference: Several studies show that web referencing patterns are non-uniform and some Web objects are more popular than others. LFU is seen to perform better in such scenarios since it considers the frequency of access while making eviction decisions.
6. Turnover: The set of objects that users are currently interested in is dynamic. Such inactive objects should be removed from the cache to make space available for new objects that are now in the active set. LFU performs badly in such workloads since it discriminates against removing frequently accessed pages, even after they have not been accessed for a long time.

We considered multiple URL requests, along with the data-size that was being requested. For instance, a user could request 40 bytes of google.com and 400 bytes of google.com. If request #1 is made before request #2, we fetch 40 bytes from the URL and add it to the cache. When the second request comes in, we find that although we fetched some data from the URL, we don't have all the requested data. This results in a miss, and new data is fetched from the URL. The old entry for google.com(with 40 bytes) is evicted and the new entry is inserted. On the other hand, consider a scenario where request #2 for 400 bytes is made before the request for 40 bytes. In this case, the first request results in a cache miss and 400 bytes are brought into the cache. When the request for 40 bytes is then made, we already have sufficient data to serve this request from cache, and it is considered a cache hit. Thus, our data load consists of tuples of {site, size} that help us mimic the scenario above.

Four different sets of workloads were generated with small(200 tuples- uniform and random) and large(500 tuples- uniform and random) data-size. These were used to test the

performance of each of the implemented caching algorithms for varying loads. These workloads were created to model random and uniform distribution of URLs from a set of 150 URLs, using python modules for uniform and random distribution. Some of the common patterns included in the workloads are:

1. A-B-C-D-E-F-G-H-I-J..: This pattern shows no repetition of page accesses.
2. A-B-C-D-E-F-G-H-A-A-A-I..: This pattern shows repetition of one page, across time.
3. A-A-A-A-B-C-D-E-F-G-H-I..: This pattern shows repetition of one page, in the beginning which is not accessed later.
4. A-A-B-B-C-C-D-D-E-A-…: This pattern shows repetition of page accesses, but without the need for any eviction.
5. A-B-C-D-A-B-C-D-..: This pattern shows repetition of page accesses in a loop, where loop size is just greater than the cache size (3 in this case).

Experiment Description

The experiments were run across local network, with the client being Macbook and the server being run on a Ubuntu 14.04 VM. Cache size is varied from 8 MB to 64 MB. Cache hit rate and average latency were measured for the four workloads described above. Cache hit rate is calculated by keeping track of the number of cache hits and total number of requests. Average latency is calculated by measuring the response time for serving requests on the server side.

We expect the following hypotheses to work:

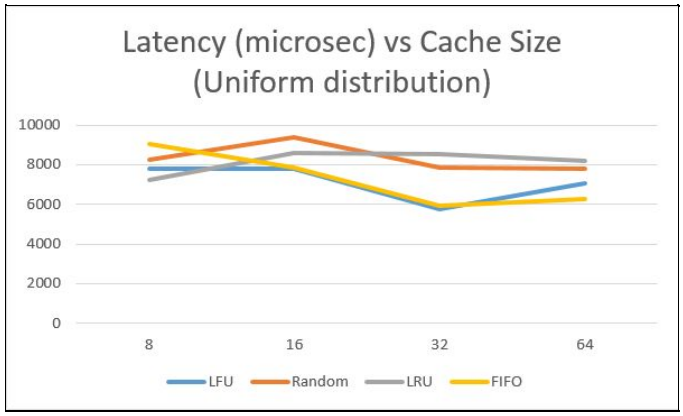
- Random caching forms the baseline for our experiments, since it involves no intelligence in cache eviction mechanisms.
- We expect that none of the caching policies perform especially better than others in case of patterns with no repetition of url accesses (A-B-C-D-E-F-G-H-I-J).
- For patterns with repetition of one url across time (For example: in case of a particularly popular pages like google.com), LFU should perform considerably better than all other caching policies.
- In case of repetition of one page in the beginning which is not accessed again later, LFU still continues to evict newer pages, which might get accessed again soon instead of evicting an older page that was accessed more frequently long back. This degrades the performance of LFU. In such scenarios, LRU performs better.
- However, LRU cache with size = N performs badly in case of loops where pages get repeated after N+1 accesses. Each page is evicted by LRU just before it was needed again. In this case, random performs better than both LRU and LFU.
- LIFO/FIFO are expected to perform the worst, specially with increasing cache sizes owing to Bélády's anomaly.
- Otherwise , hit ratios must increase with the increasing cache sizes.
- Increasing the total number of URL request should also increase the hit-rate, owing to a better chance of finding an entry in cache.

- In all cases, the proxy server caches should perform better in terms of latency as compared to direct internet access mechanisms.

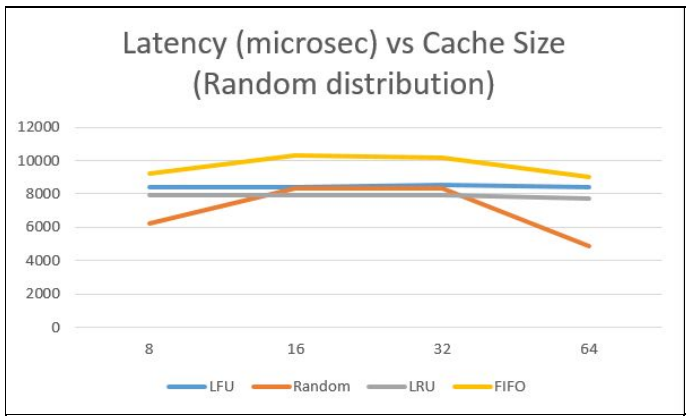
Experimental Results

Please find below graphs summarising the results of our experiments:

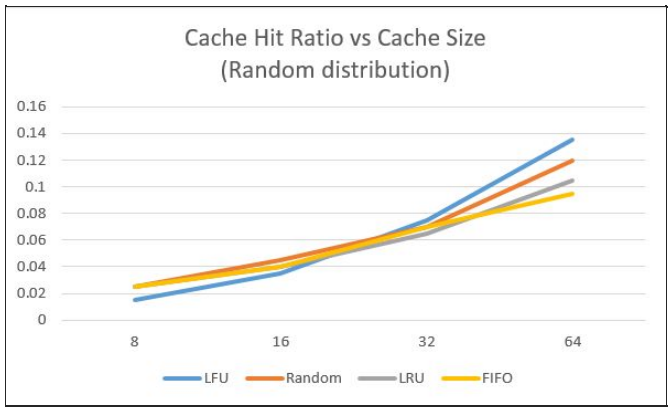
1. The following graph shows the average latency of all the requests of uniform distribution with varying cache size. We see that as expected LFU generally perform favorably than FIFO and Random, although LRU performs worse probably because of particular access patterns.



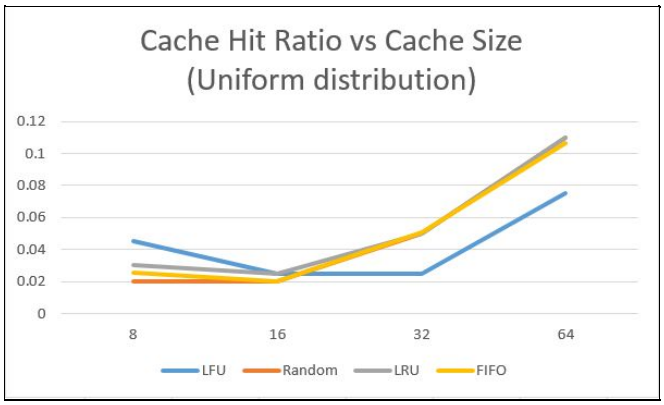
2. The following graph shows the average latency of all the requests of random distribution with varying cache size. We see that as Random Cache performs better than its counterparts, closely followed by LRU and LFU. FIFO performs worst as expected.



3. The following graph shows the cache hit ratio of all the requests of random distribution with varying cache size. We see that all caching mechanism perform almost equivalently under random workload, since there is no particular pattern involved. FIFO performs worst as expected. Also we see that the hit ratio steadily increases with the increase in cache size, as expected.



4. The following graph shows the cache hit ratio of all the requests of uniform distribution with varying cache size. LFU unexpectedly has a lower hit ratio than its counterparts, and we believe that it is because of large size of documents requested and constant evictions in LFU. FIFO and Random due their simple eviction patterns perform better. Also we see that the hit ratio steadily increases with the increase in cache size, as expected.



Conclusion

In this project, we implemented and managed multiple caching mechanisms in a proxy server. We conclude that different caching mechanisms perform better for different workloads or request patterns, hence a combination of these might be good in a practical setting. LFU generally performs the best given a uniform workload, LRU closely following it. Random distribution workload has unpredictable random outcome but generally more favorable than LIFO/FIFO mechanisms. Caching mechanisms like LRU-min are advanced policies which also aim at increasing the overall efficiency of the application and minimize the number of documents replaced. In general, it is observed that cache performance increases as cache size increases.

References

1. http://www.hpl.hp.com/techreports/98/HPL-98-97R1.pdf
2. http://www.webabode.com/articles/Web%20Caching.pdf

3. http://webpersonal.uma.es/~ECASILARI/Research/Papers/Congresos/2006/Melecon_francis_p210.pdf
4. http://www.webabode.com/articles/Web%20Caching.pdf
5. https://en.wikipedia.org/wiki/Cache_(computing)
6. https://en.wikipedia.org/wiki/Page_replacement_algorithm