CS 6210 ADVANCED OPERATING SYSTEMS

PROJECT 3

RPC-BASED PROXY SERVER

Team Members:

Anouksha Narayan (anarayan33@gatech.edu)

Nikita Mundhada (mundada.nikita@gatech.edu)

1. Introduction

Remote procedure call (RPC) is a powerful technique for constructing distributed client-server applications. The main idea behind this is that one can extend the concept of simple local procedure calls to scenarios where the called procedure does not necessarily lie in the same address space as the calling procedure. By using RPC, programmers of distributed applications avoid the details of the interface with the network.

RPC is similar to a function call. Like a function call, when an RPC is made, the calling arguments are passed to the remote procedure and the caller waits for a response to be returned from the remote procedure. Figure 1 shows the flow of activity that takes place during an RPC call between two networked systems. The client makes a procedure call that sends a request to the server and waits. The thread is blocked from processing until either a reply is received, or it times out. When the request arrives, the server calls a dispatch routine that performs the requested service, and sends the reply to the client. After the RPC call is completed, the client program continues. RPC specifically supports network applications.

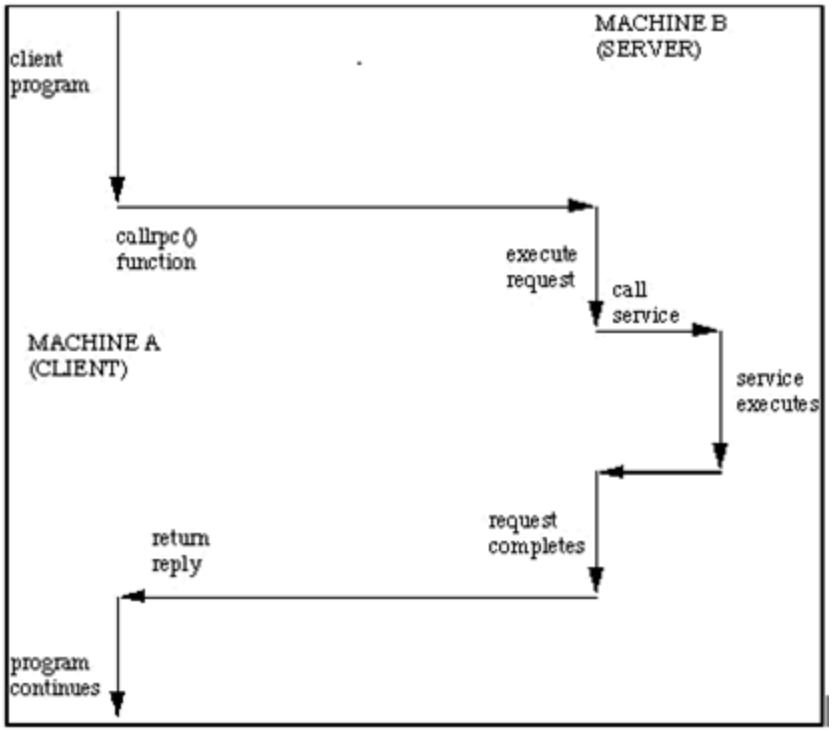

Fig. 1 Remote Procedure Calling Mechanism

For this project, we built a simple web proxy server using Apache Thrift to implement different caching policies, while evaluating the performance of each policy under

different load conditions and using different caching schemes. For each policy, we measured the average response time and hit rate over varying cache sizes and different workloads.

2. Caching Policies

In our code, the main cache is implemented as a map (A map is internally implemented as a binary search tree by CPP). For our map, the URL is the key and the corresponding page content is the value for all key-value pairs in the map. The advantage of using a map is that the search complexity is $O(\log_2 N)$ where 'N' is the size of cache as a map is nothing but a binary search tree.

For the different caching policies, various other kinds of data structures are used which store the URLs from the cache along with any metadata that may be required when evicting pages using a particular page replacement policy.

To implement the random caching policy, along with the cache map, another vector is maintained which contains all the URLs presently in the cache. Insertion into cache using this policy is thus of the complexity O(1) as insertion into vector using push_back()Linear on the number of elements erased is also of complexity O(1).
In order to evict the corresponding URL along with its data, when a random number is chosen, the URL corresponding to that position in the vector is then chosen and consequently, that URL is evicted from the cache. This has a complexity of $O(1) + O(\log_2 N)$ i.e. $O(\log_2 N)$ since we are removing only a particular element from the vector which dynamically adjusts itself (unlike an array data structure) which has a complexity O(1) and then removing the corresponding page from the cache which has complexity $O(\log_2 N)$.

For the Largest Size First caching policy, there are three additional data structures that are maintained. One is a struct which contains the URL name and the corresponding size of the page. Second is a vector of the previous struct data type and the third is a heap which is constructed from this aforementioned vector of structs. The construction of heap has complexity $O(\log_2 N)$. Since the max-heap data structure automatically ensures that the root is the page having the largest size, evicting a page from the cache has a complexity of $2O(\log_2 N)$ i.e. $O(\log_2 N)$ since

deleting an element from max-heap is of complexity $O(\log_2 N)$ and deleting from cache is of complexity $O(\log_2 N)$.

To implement the First In First Out (FIFO) caching policy, a queue data structure is maintained which contains the URLs presently in the cache. By its very nature, a queue data structure stores data in the order in which they arrive i.e. the first element to enter the queue will also be the first element to leave the queue. Insertion into queue is of order O(1). Thus, page eviction becomes extremely simple since we only have to call the pop() function on the queue which removes the first URL that had come in and subsequently evict that page from the cache as well. Again, this policy has a time complexity of $O(1) + O(\log_2 N)$ i.e. $O(\log_2 N)$.

3. Cache Design

Following are the different caching policies that we implemented for page eviction:

a) No Caching: Our very first test case was to evaluate the performance of the system with no cache present at the server side. The average response time for this case was much higher than the following cases since, for each request, due to the lack of a local cache, the server had to fetch the data from the Internet instead of just performing a cache lookup.

b) Random: For this policy, a page would be chosen at random from the cache and then evicted to make room for a new request that has come in. This eliminates the overhead cost of tracking page references. Usually it fares better than FIFO, and for looping memory references it is better than LRU (Least Recently Used), although generally LRU performs better in practice. This process of randomly choosing a page and evicting it continues until there's enough place in the cache to accommodate the new page. This policy is unpredictable in the sense that it's difficult to say which page can be chosen for eviction. This also makes it fairer as there's an equal probability that a page will be chosen for eviction. However, if a proper random generator isn't used then it's possible that certain kinds of pages will be removed all the time. Also, it does not make optimum use of the cache size when evicting a page from the cache.

c) Largest Size First: As the name suggests, this caching policy looks for the page with the maximum size in the cache and evicts that to make room for the new page

coming in. Again, this process of finding the maximum size and then evicting continues until there's enough place in the cache to accommodate the new page. This caching policy works well under the assumption that users are less likely to re-access large documents because of the high access delay associated with such documents. Implementing this policy generally leads to higher cache hit rates since you are clearing off the big pages in the cache to make room for multiple medium-sized pages. Thus, response time will also be much better. However, there may be cases where the large pages are frequently accessed by the user and hence must be stored in the cache for a longer time, which becomes difficult with this page replacement policy.

d) First In First Out (FIFO): The simplest page replacement algorithm is the FIFO algorithm. According to this algorithm, pages are evicted from the cache in the order in which they arrive i.e. the first page to enter the cache will also be the first page to be evicted from the cache. 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 which is a phenomenon where increasing the number of page frames increased the page faults (cache misses) that occur.

4. Metrics for Evaluation

The metrics used for evaluation are response time and cache hit rate. The response time is the time taken for the proxy server to retrieve the HTML page content. It is averaged across all requests including the cache hits and misses. The cache hit rate is the number of hits divided by the number of total cache accesses for the requests in the workload. The performance of the proxy server is measured using these metrics by varying the maximum cache size.

5. Workloads

We created three different workloads from the same list of URLs.

a) Workload with repetition: The URL list was simply replicated. So, the workload essentially had double the number of URLs as the URL list i.e. it consisted of two identical sets of URLs in a random fashion. So if the URL list is "abcdd". The workload is "abcddabcdd"

b) Workload with repetition then sorting: We took the workload with repetition described above and sorted the list of URLs alphabetically in ascending order. This automatically ensured that duplicate URLs in the list were sent as consecutive requests to server. So if the URL list is “dbac”. The workload is “aabbccdddd”.

c) Workload ordered by page size and then repeated: We ordered a non-repeating set of URLs in ascending order based on the size of the page. Hence, the requests taking less time would go first to the server followed by the requests requiring more time to load This ordered list was then repeated in its entirety twice to make up the final workload.

6. Experiment Description

The workload was read from a file on the client and the web pages corresponding to each URL were requested one by one from the server. We tested each workload with each policy, namely FIFO, Random and Largest Size First (MAXS). We also tested each combination of policy and workload with cache sizes of 0(no cache), 32KB, 64KB, 128KB, 256KB and 512KB. So we had a total of 2 workloads* 3 policies * 6 cache sizes = 36 test cases.
Untitled documentUntitled document
7. Experimental Results and Analysis

a) Workload with repetition:

The graph below shows the average response time performance of the different caching policies for varying cache sizes:

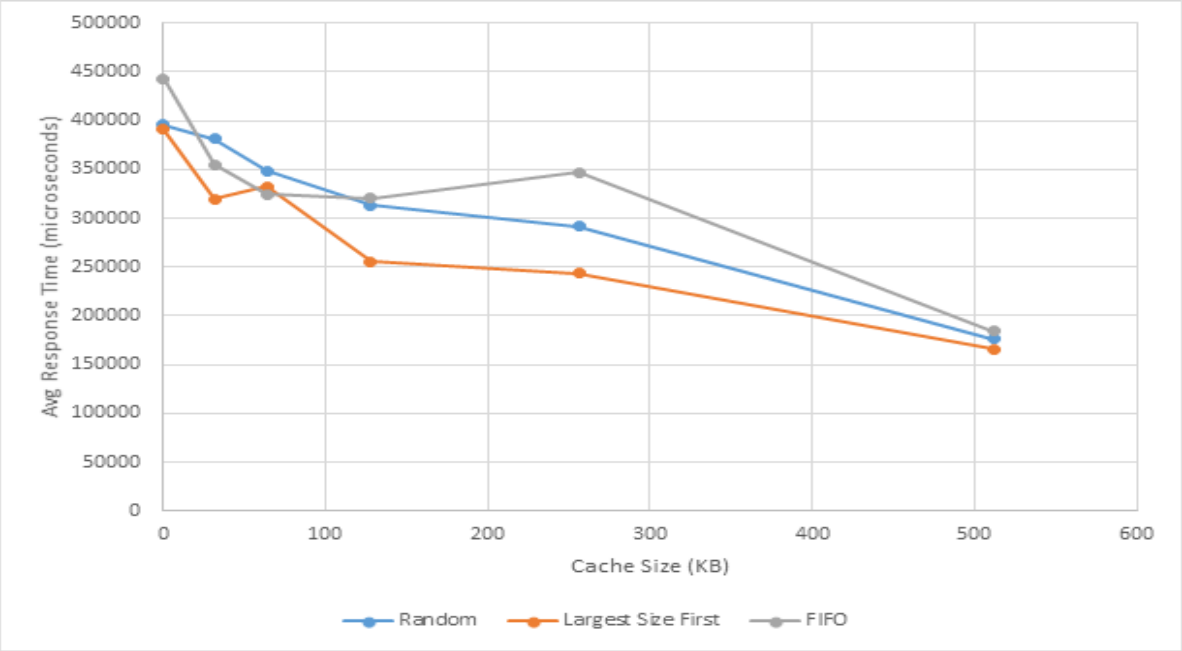

Fig. 2 Time Performance of the Caching Policies vs Cache Size

As depicted in the graph above, the Largest Size First policy performs better than the other two since removing the largest page makes more room in the cache for other pages, thus reducing the response time since most of the pages requested by the client would be in the server's local cache. This is especially true for the random workload we constructed since URLs are repeated in a random fashion. Random page replacement takes more time than Largest Size First but does much better than FIFO since each page is fair game for eviction. FIFO performs poorly in comparison to the other two policies since each URL is evicted from the cache turn by turn and there's no optimization based on the available cache size.

The graph below shows the hit rate performance of the different caching policies for varying cache sizes:

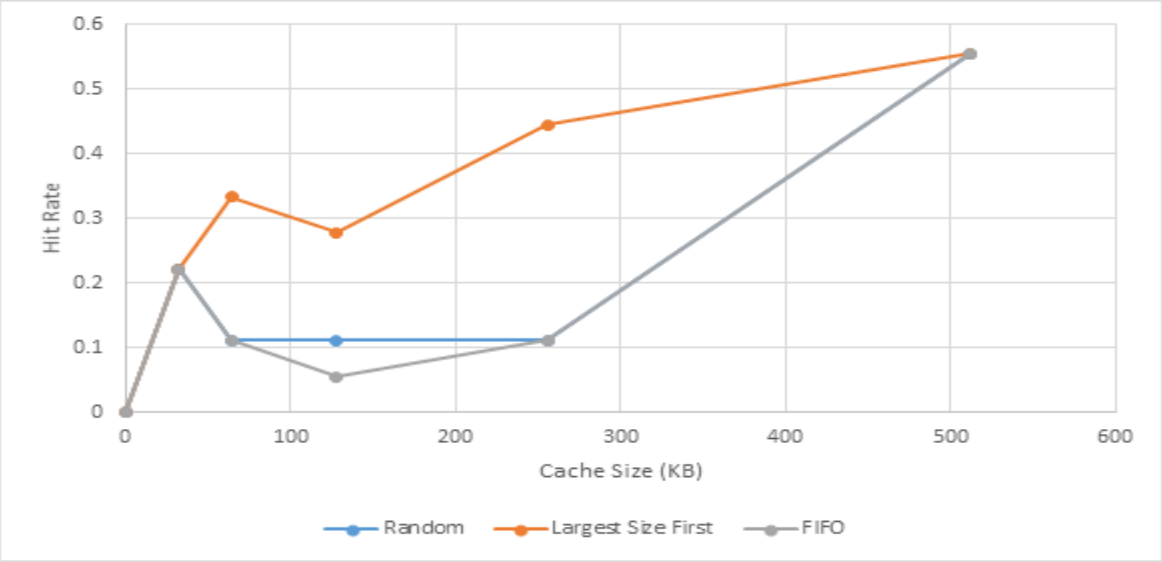

Fig. 3 Hit-Rate Performance of the Caching Policies vs Cache Size

Again, the Largest Size First policy has a much higher hit rate than the other two policies. Since more pages can be stored in the cache, more are the number of hits when repeated URLs are requested (as occurring in this workload). FIFO has the lowest hit rate here while Random does slightly better than FIFO but not as good as Largest Size First. As mentioned before, FIFO also suffers from Bélády's anomaly where increasing the number of page frames increases the page faults or cache misses that occur (thus reducing the cache hit rate).

b) Repeated then sorted workload

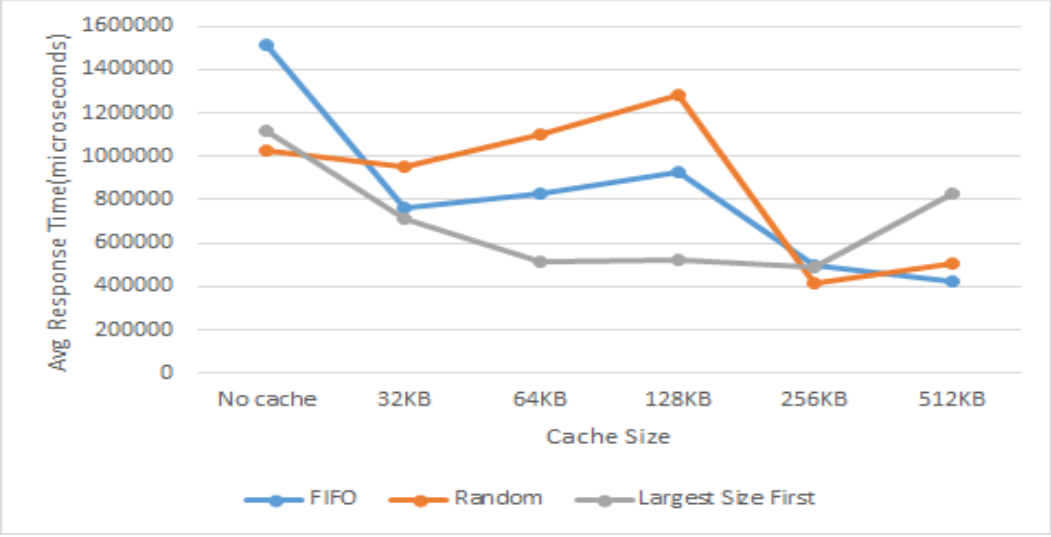

Fig. 4 Time Performance of the Caching Policies vs Cache Size

As can be seen in the figure, the response time for Largest Size First algorithm is much better than FIFO and Random. This is because, in general, FIFO or Random evict more pages than Largest Size First. So, there is a higher chance of a page being found in the cache (assuming that higher weight requests are infrequent) as compared to the other two. The response times of FIFO and Random are comparable, with Random performing slightly better. The workload consists of two identical set of URLs and when we sort it, similar URLs from both sets get clubbed together. So, a page is likely to be in the cache when it is retrieved the second and subsequent times.

Also in general, the response time of requests reduces with the increase in cache size. Larger cache size means fetching of more web pages from the cache as compared to the internet. Hence, the response time also reduced because the number of cache misses reduced. This is why the poorest performance is for no cache.

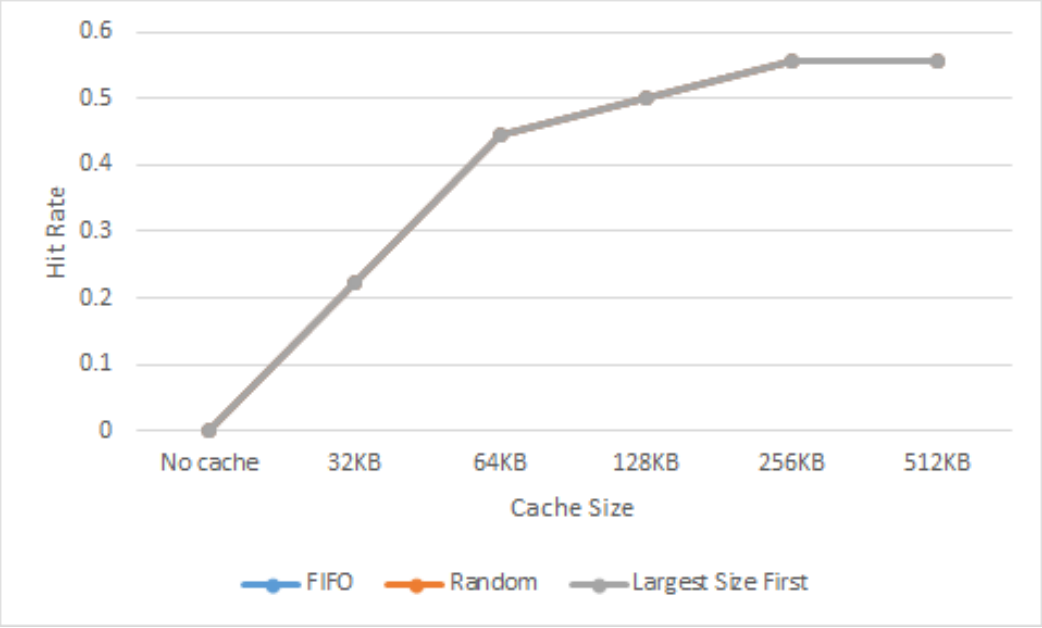

Fig. 5 Hit-Rate Performance of the Caching Policies vs Cache Size

The hit rate improves with an increase in cache size as more pages can be accommodated in cache as can be seen in the graph. The hit rate is the same for all three cache replacement algorithms for the same cache size. This is because, as the

workload is repeated then sorted, there would be a cache miss the first time a URL ‘A’ is requested from the server, but subsequent requests for ‘A’ would all be cache hits. So, the number of cache hits and cache misses would be exactly the same no matter what cache replacement policy is used there.

c) Workload ordered by page size and then repeated:

The graph below shows the average response time performance of the different caching policies for varying cache sizes:

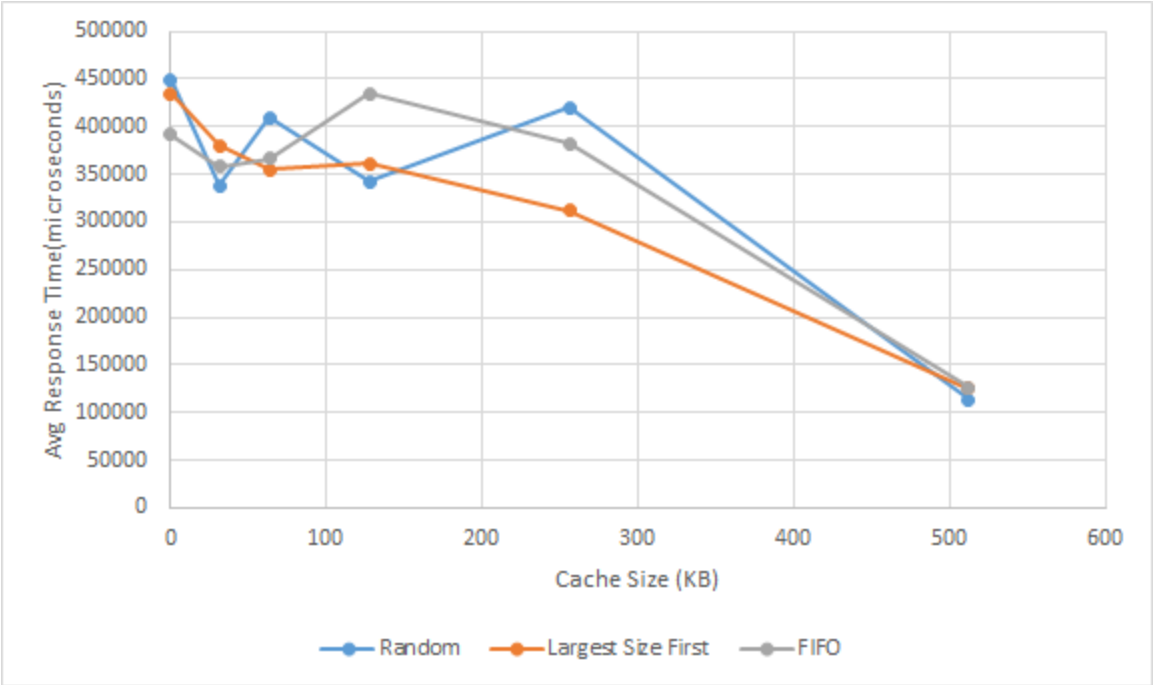

Fig. 6 Time Performance of the Caching Policies vs Cache Size

As shown in the graph above, Largest Size First again performs better than the other two caching policies. This is because more room is created in the cache when the larger pages are evicted, thus reducing the time taken by the server to fetch a page since a copy of it would exist in the local cache. FIFO on an overall basis takes more time than the other two policies for this workload. The reason behind this is that since the pages are ordered by size, when a request is made for a large page, FIFO would nearly empty the cache (now containing the smaller pages) to make room for the larger pages. Thus more time is spent in clearing the cache leading to poorer performance.

The graph below shows the hit rate performance of the different caching policies for varying cache sizes:

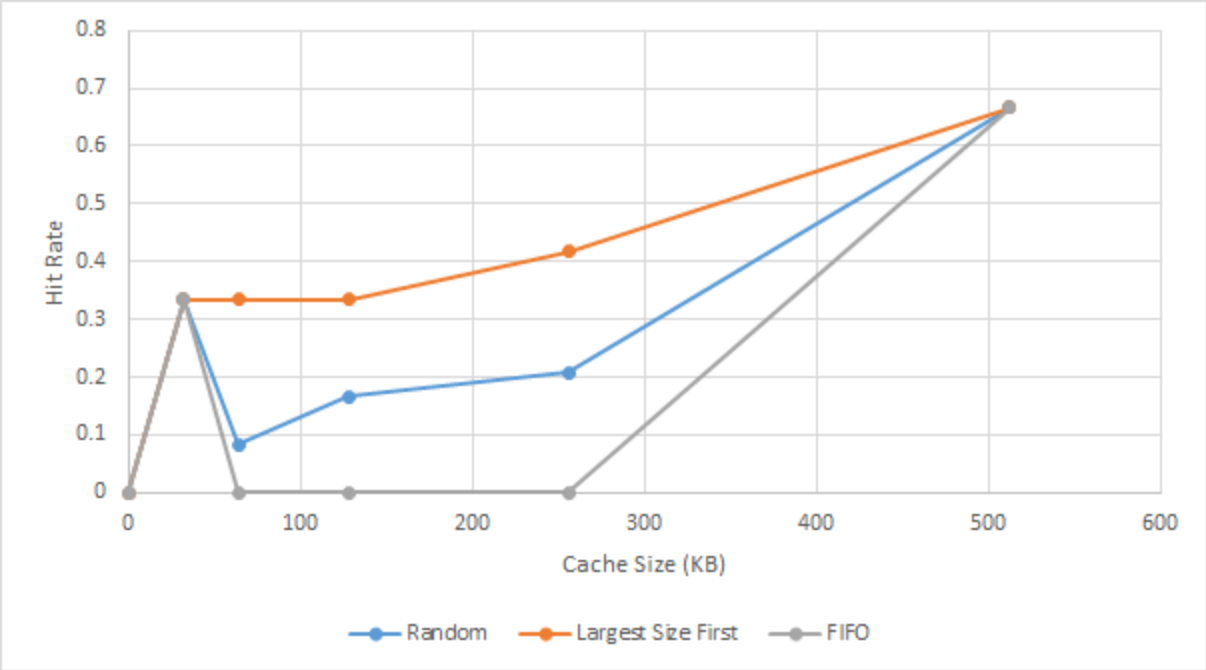

Fig. 7 Hit-Rate Performance of the Caching Policies vs Cache Size

Again, the Largest Size First policy has a much higher hit rate than the other two policies. Since more pages can be stored in the cache, more are the number of hits when repeated URLs are requested (as occurring in this workload). The Random policy performs better than FIFO but is not as good as Largest Size First. Since random pages are chosen for eviction, it is difficult to predict how many cache hits or misses may occur. However, its performance will always be better than FIFO. An interesting point to note here is that the hit rate for FIFO becomes 0 for several cache sizes (as can be seen in the figure above). This occurs due to the arrangement of pages (according to size) in the URL list. Since the pages are ordered by size, when a request is made for a large page, FIFO would nearly empty the cache (now containing the smaller pages) to make room for the larger pages. Thus, when URLs are requested again, it would always result in a cache miss and hence, a hit rate of 0.

8. Conclusion

In general, out of all the three algorithms, FIFO performs the worst. The hit rate increases with an increase in cache size and the response time reduces with increase in cache size.