CS6210 – Project 3
RPC-Based Proxy Server

Project Report

Team members:

Dhananjayan Ramesh
Prabu Shyam Mayavaram Mahalingam

Introduction:

Remote Procedure Call (RPC) is a protocol that a program can use to request a service from another program located in another computer in a network without having to understand the network details. RPC uses the client/server model where the requesting program is a client and the service-providing program is the server. RPCs are a powerful and commonly used abstraction for constructing distributed applications.

RPC - Step-by-step:

1. The client makes a local procedure call to the client stub with parameters pushed to the stack.
2. Marshalling: The client stub packs the parameters into a message and makes a system call to send the message.
3. The client's local operating system sends the message from the client machine to the server machine.
4. The local operating system on the server machine passes the incoming packets to the server stub.
5. Unmarshalling: The server stub unpacks the parameters from the message.
6. Finally, the server stub calls the server procedure and the reply traces the same steps in the reverse direction.

We have implemented an RPC based proxy server with caching mechanism using libcurl where a client can request a URL and a client application to fire requests to the server for analyzing the performance of the server. The cache is implemented as a limited in-memory cache with the replacement policies - LRU, LFU and RAND.

Proxy Server:

The Sun RPC based web proxy server uses a cache data structure to cache html pages based on the requested client URLs and on successive requests, the cached content is returned. Since the user can request various URLs, in varying number of times and request rates, and the cache size is limited, the replacement policy plays an important role in saving the network bandwidth and reducing client latency.

Cache Replacement Policies:

Cache replacement policy plays an important role in determining the performance of the proxy server; however the performance of every policy depends on the type of input workload. The same policy might result in the best and worst performances for different types of workload. Hence, no policy can be termed as “best for any input”.

For this project, we have implemented the below three most common types of replacement policies.

Least Recently Used:

In this policy, the assumption that users tend to access recently visited URLs more often is exploited. A visited list of URLs is maintained in the order of page visits. The most recently accessed URL is placed at the tail of the list, and the least recently accessed URL at the head. On a cache hit, the cached URL existing anywhere in the list is replaced to the tail end.

Data structures :

To avoid linear search in all possible scenarios(to cache an entry, lookup an entry, lookup the address of the cache block, lookup the position in the list), we used three separate data structures,

a) a hashtable to do a constant time lookup, if the URL string has been already processed by the server

b) a cache of a large number of blocks, each block with a variable content size to store the URL and it's corresponding HTML.

c) a linked list of nodes to maintain the LRU order of page visits.

We use a hashtable to quickly identify the index location of a string URL. The hashcode of a given input URL is computed using a DJBHash algorithm and the resulting code is reduced(modulo) to the number of entries in the hashtable. Every item stored in the hashtable contains the URL, a flag to know if the URL is present in the cache, if so the index to the corresponding cache entry and the pointer to the node in the LRU list.

The cache data structure is a static array of an arbitrarily large size. Every cache block can store an URL and a variable length of the corresponding HTML page content. The total cache size is limited so that the accumulated size of the all the entries in the cache is always less than or equal to the total cache size.

The LRU list is a priority queue ordered based on the page visit time, with the head pointing to the least recently accessed page to be first candidate to be evicted and the tail pointing to the most recently accessed page. Every node in the LRU list also contains an index to the cache block to quickly lookup and process it.

Algorithm:

On receiving a web page request, the remote procedure first computes the hashcode of the input URL and checks if the URL had been already processed and present in the cache. If the hashtable entry points to an existing cache block for the URL, the stored HTML content is retrieved from the cache data structure. The corresponding node in the LRU list is pushed to the tail end to indicate that it has been recently accessed.

If an existing cache block entry is not found, then the CURL HTTP request is performed to retrieve the page. On completion of the HTTP request, the obtained HTML page has to be cached. But since, the free space in the cache might not be enough to fit the entire HTML page or the total no. of existing cache entries might be equal to the total no. of possible cache entries, at this time the cache occupancy status is evaluated. If the size of the free cache space plus the space required by the new HTML content exceeds the cache capacity, some of the entries need to be evicted from the cache.

The entry at the head of the LRU list is first evicted, and the process is continued until the free cache space is sufficient to store the new HTML content. Also, when a cache entry is evicted from any location of the contiguous block, the free block is moved to the end and the pointers updated appropriately. This negligible overhead avoids a linear search for a free cache block.

The new cache entry is placed at the tail end of the LRU list and the total cache occupancy is updated to include the newly cached web page.

Least Frequently Used

This policy exploits the assumption that the web pages with high number of accesses are more likely to be accessed again in the future. Thus similar to LRU queue, a list of cached pages is maintained in the decreasing order of the frequency of the page visits. When a cache hit occurs, the frequency of the corresponding node in the LFU list is incremented and the links are updated to maintain the LFU order.

The LFU policy works better when the incoming request URLs conform to a lean euclidean curve in the sense that some pages are accessed more frequently than most of the other pages, and if this set of active pages fit into the cache, the LFU achieves the maximum performance.

Random Eviction

This policy implementation does not maintain a separate list as the eviction candidate is chosen at random. The random eviction policy performs averagely in most of the cases including the worst combination of the input set of URLs.

Experiments Conducted:

Platform: Ubuntu 11.1 on AMD64

The experiments were conducted to compare the performance of the LRU/LFU/Rand replacement strategies by varying the number of input URLs, and cache capacities. We considered the cache hit ratio and the service request time(latency) for evaluating the performance of the various cache replacement policies.

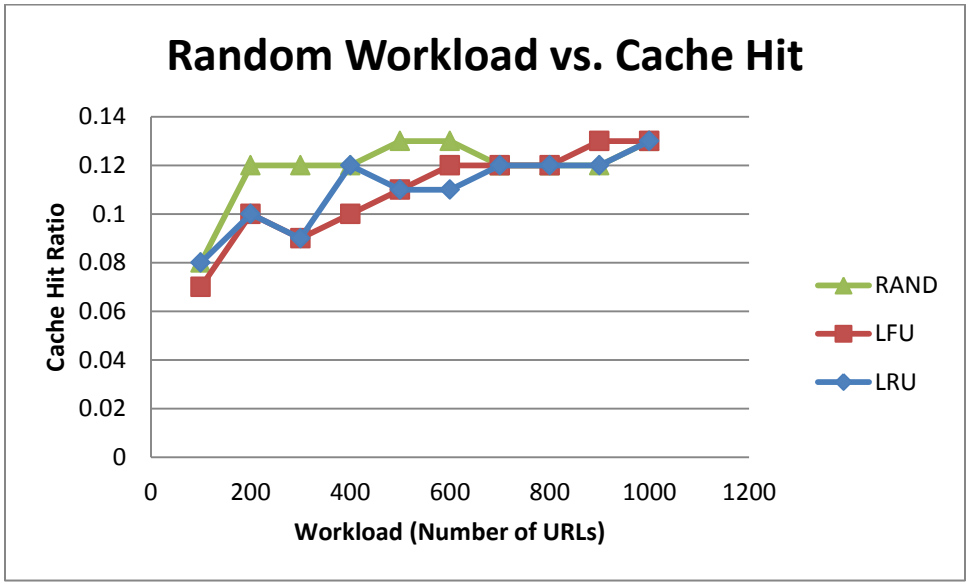
Our input URL source is the list from the 100bestwebsites.org and from this source list, we generated the below set of various workloads of sizes from 100-1000 in increment steps of 100.

1. random URLs picked from the source list
2. contiguous redundant sets(100 in each set) of URLs just as it is in the source list
3. locality based workload with arbitrary URLs appear in succession favoring LRU
4. biasing based workload in which random sets of URLs are repeated at arbitrary intervals
5. frequently repeating pattern favoring LFU

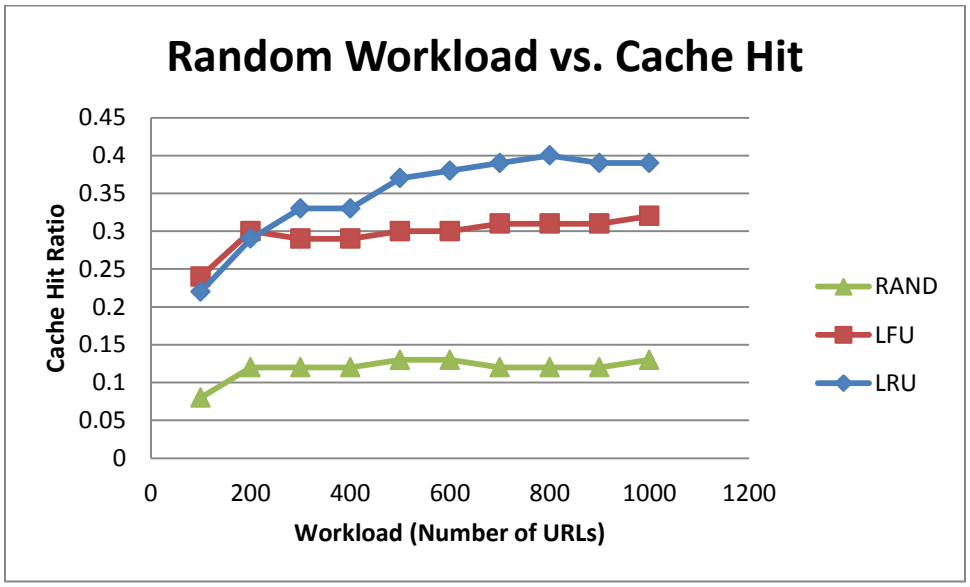
Experimental Results & Evaluation:

1. **Random URLs picked from the source list**

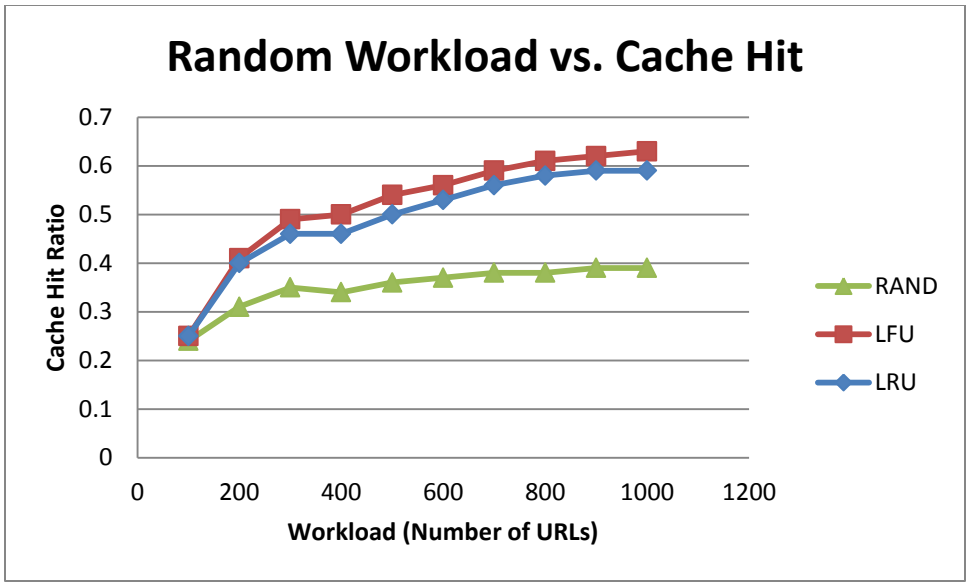
Cache Size = 1MB



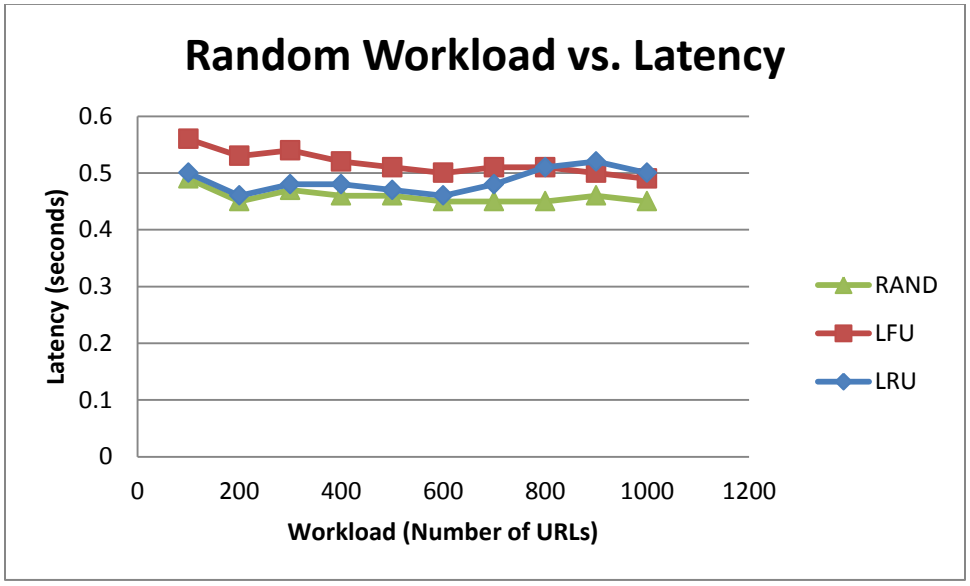
Cache Size = 3MB



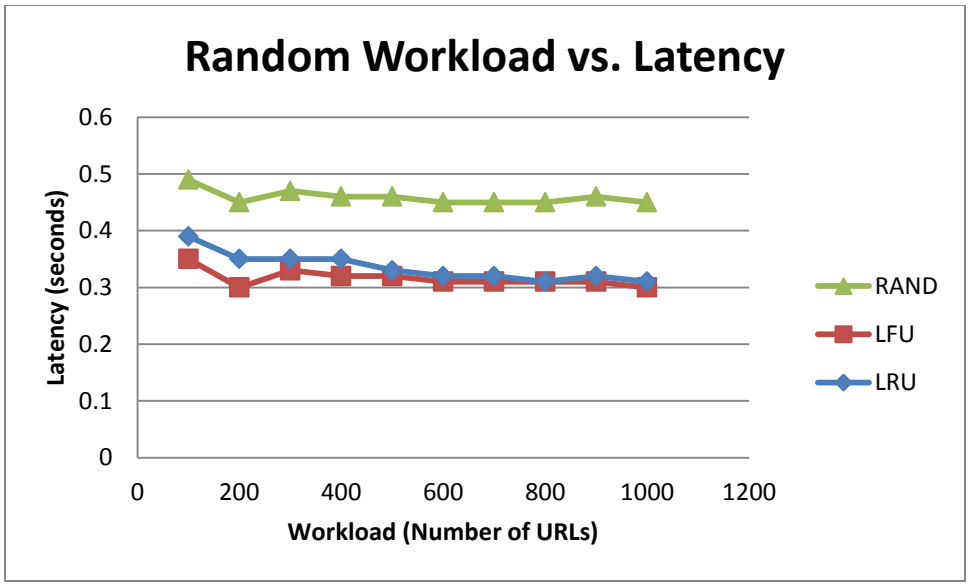
Cache Size = 5MB



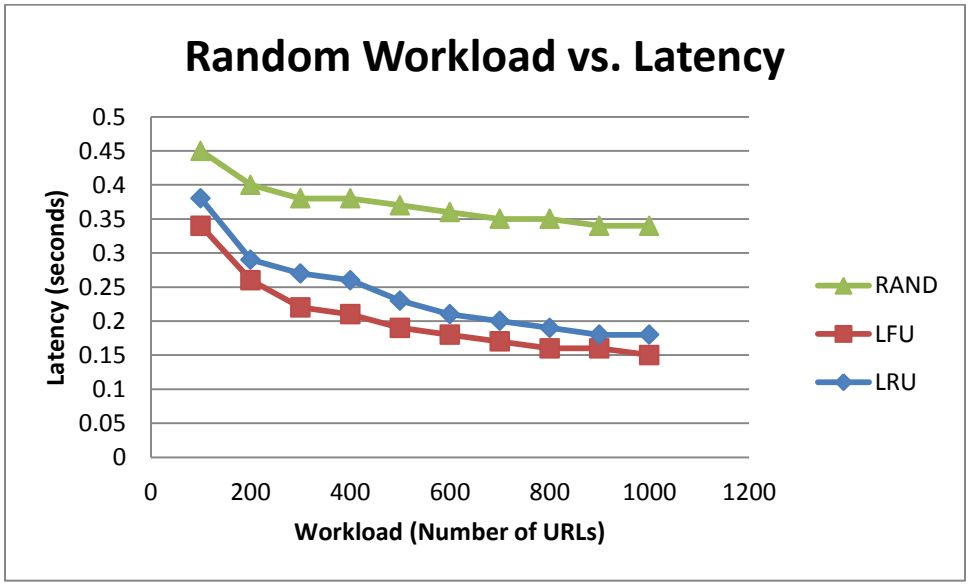
Cache Size = 1MB



Cache Size = 3MB



Cache Size = 5MB

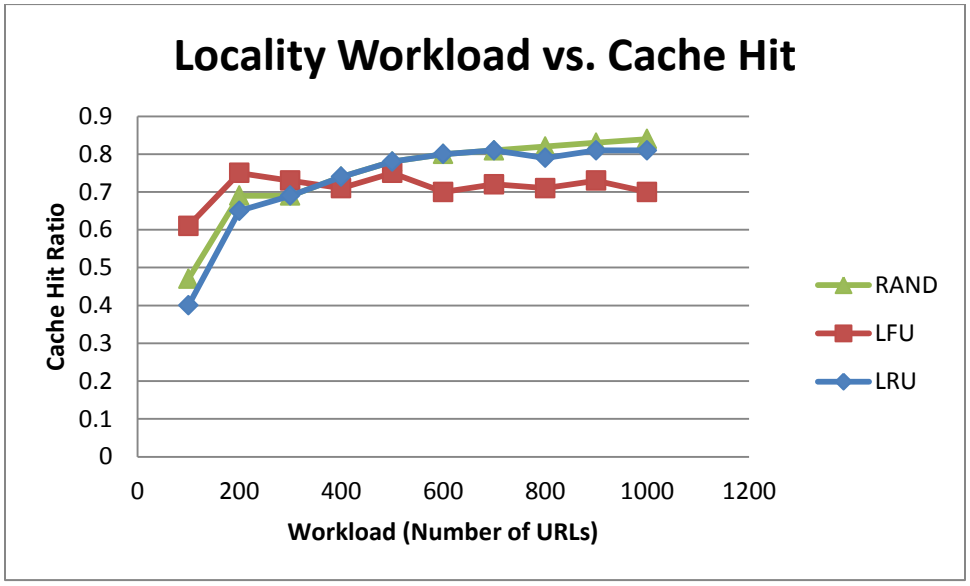


Inference:

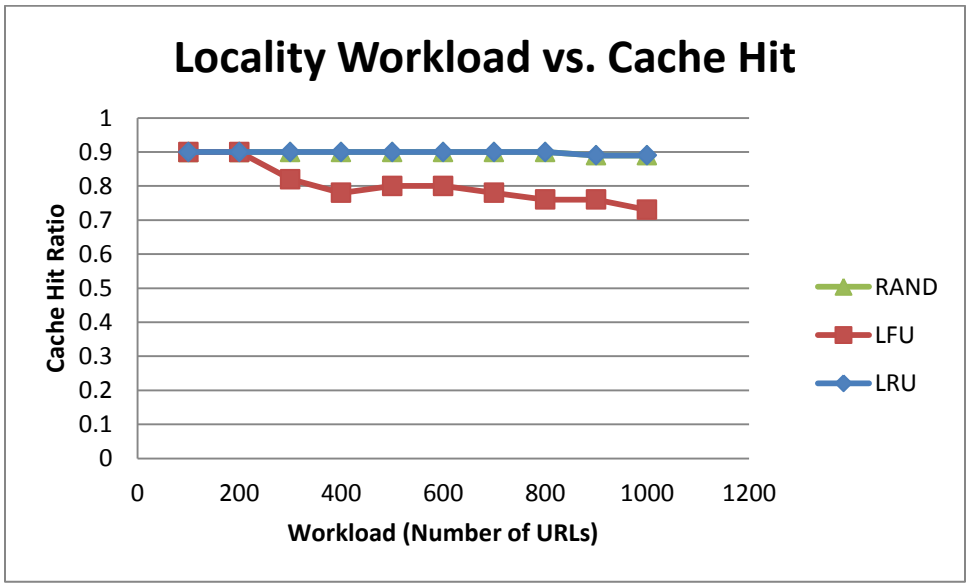
- In a completely randomized workload, we see that the random replacement policy performs moderately better than LRU and LFU algorithms in terms of hit ratio.
- As the workload increases, LFU and LRU intermittently perform better due to the randomness in the distribution of URLs.
- When the cache hit ratio improves, the latency also reduces somewhat, but since the hit ratio does not differ significantly, we only notice a minor variation in latency.
- In smaller caches, the random policy performs better compared to LRU and LFU. As the cache size increases, LFU cache hit ratio increases gradually and stabilizes, whereas the LRU cache hit ratio spikes suddenly with increase in cache size and stabilizes quickly.

2. Locality based workload with arbitrary URLs in succession favoring LRU

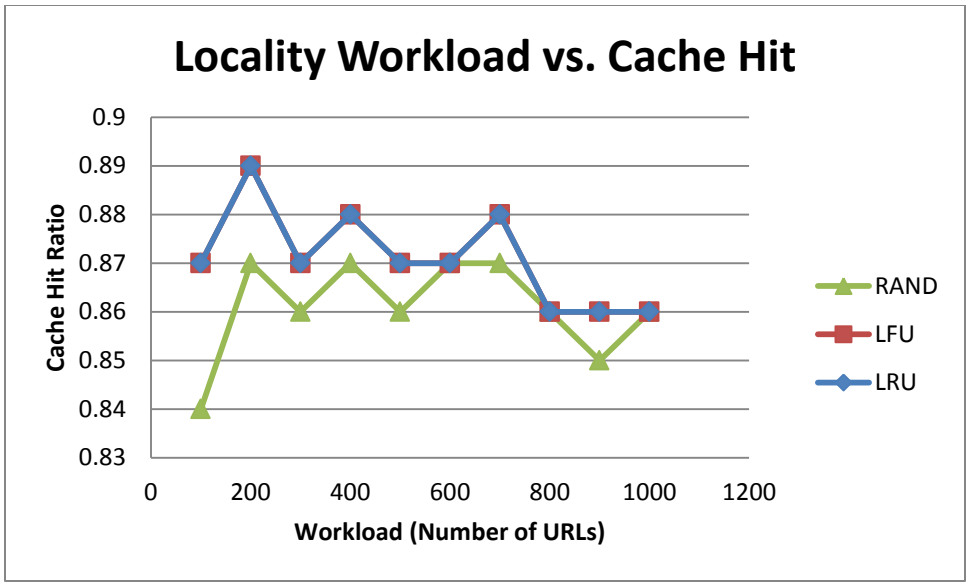
Cache Size = 1MB



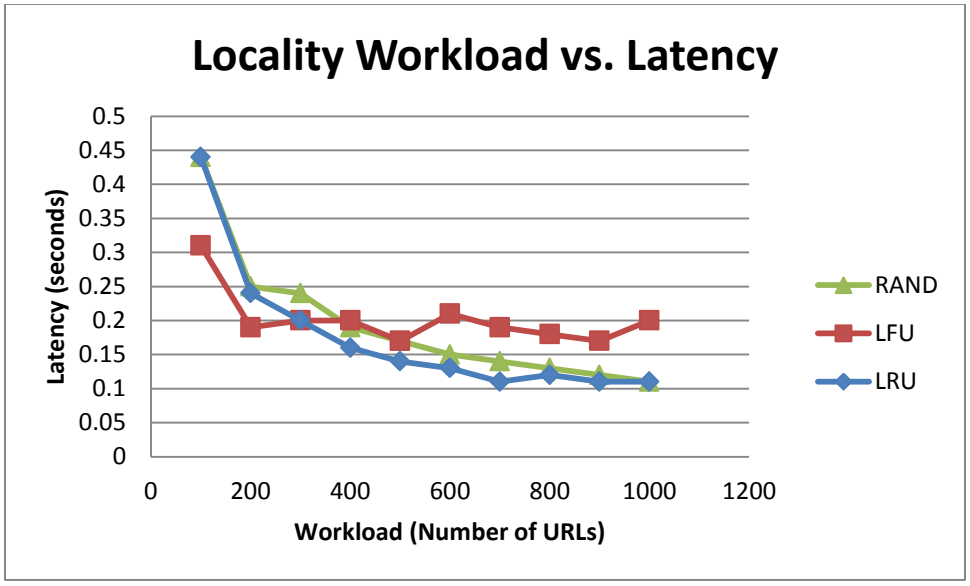
Cache Size = 3MB



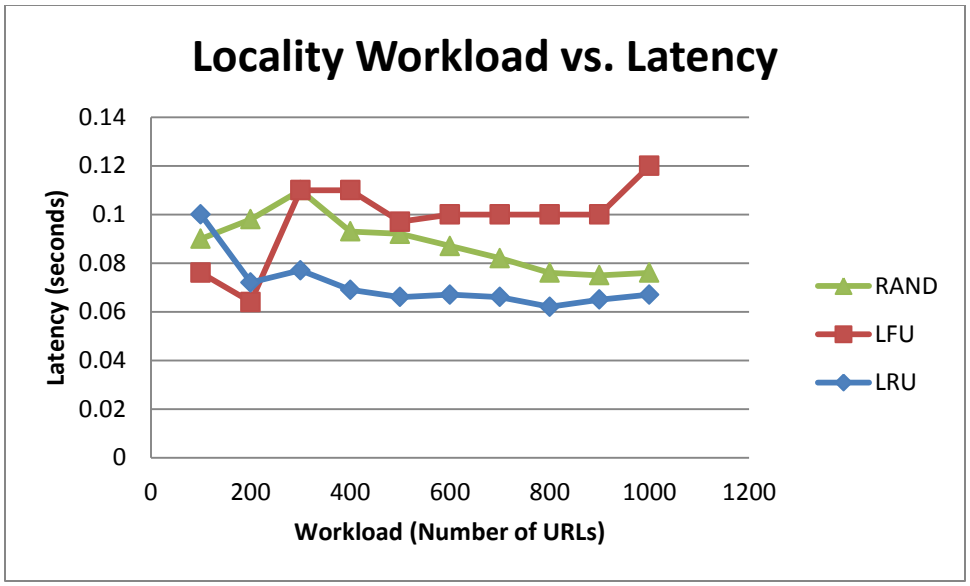
Cache Size = 5MB



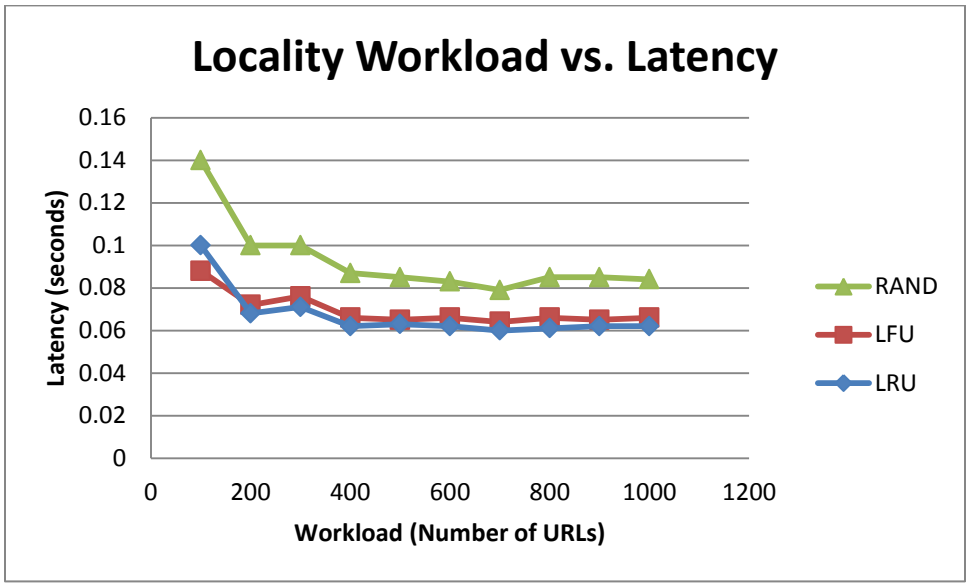
Cache Size = 1MB



Cache Size = 3MB



Cache Size = 5MB



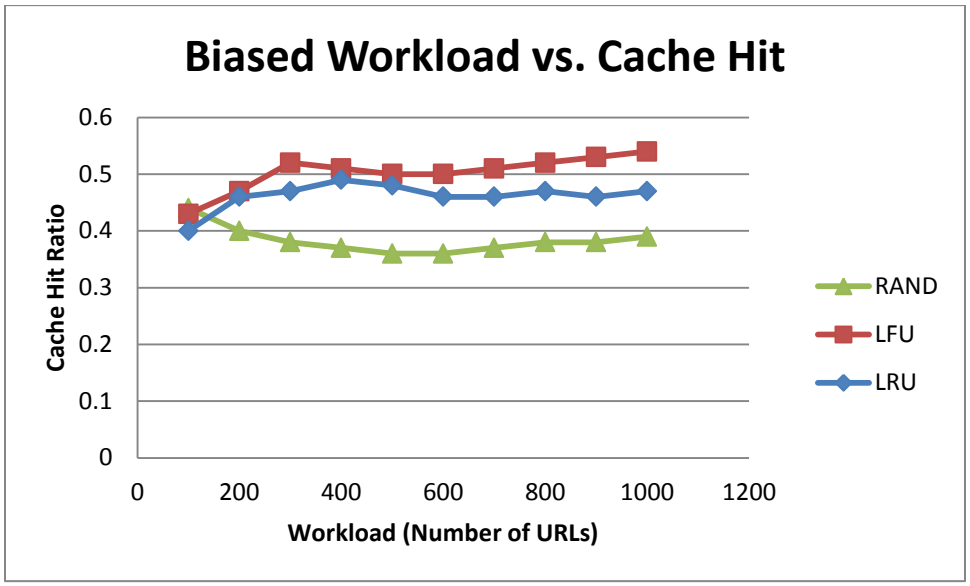
Inference:

- As expected with the pattern of the locality based workload, the hit ratio is higher in LRU compared to LFU due to the successive requests to an arbitrary URL.
- The cache hit ratio for LFU is almost stable since the repeating pattern of individual URLs does not affect the number of hits considerably.
- The size of the cache initially affects the hit ratio of the random policy with less number of workload. But, as the number increases, the hit ratio improves slightly.
- When the cache hit ratio leaps with smaller number of URLs, we see that the latency also reduces proportionately.
- As expected, the cache hit ratio stabilizes with higher cache size. Since this input pattern is designed in such a way that LRU is favored, it does not give importance to

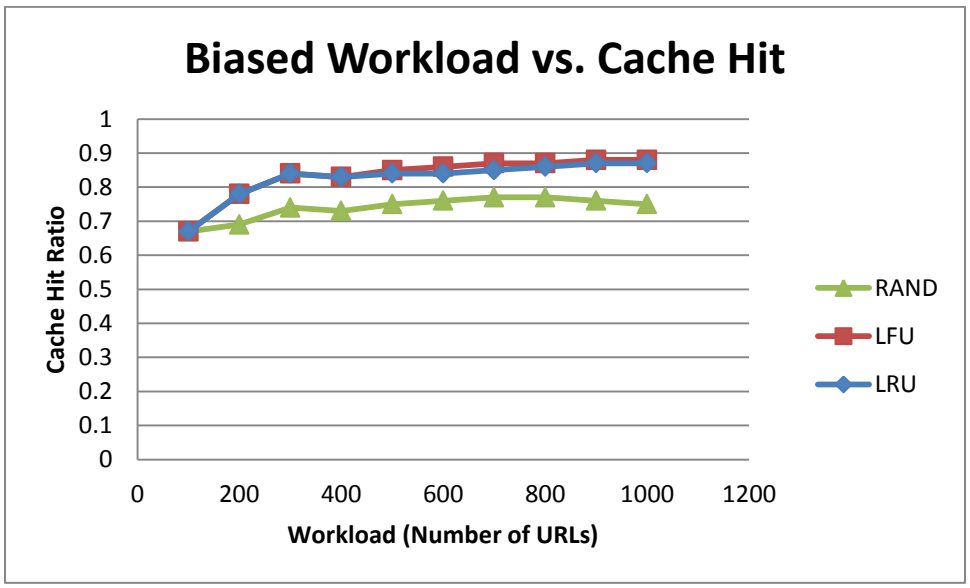
the frequency of cache hits and hence the minor drip in cache hit ratio for LFU with 3MB cache size.

3. **Biasing based workload in which random sets of URLs are repeated at arbitrary intervals**

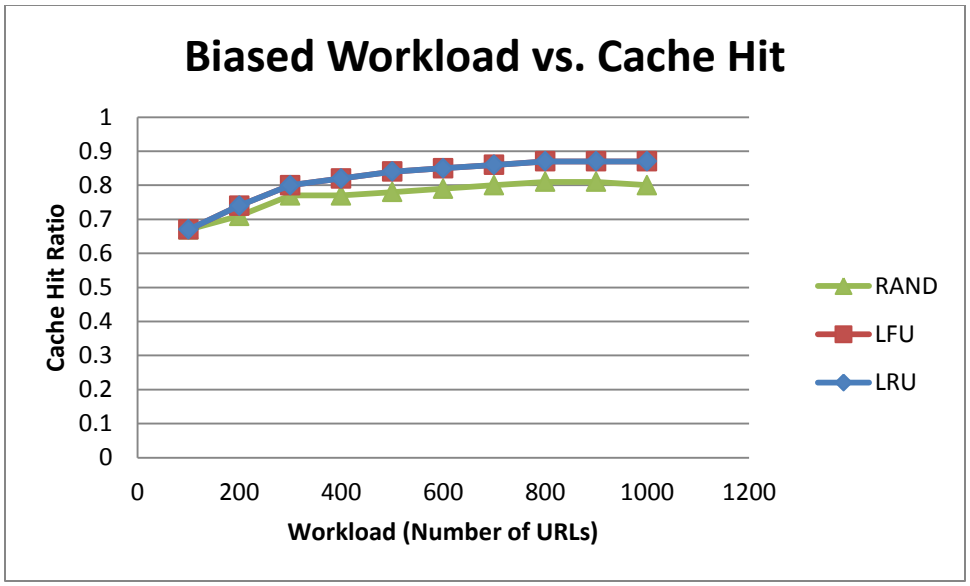
Cache Size = 1MB



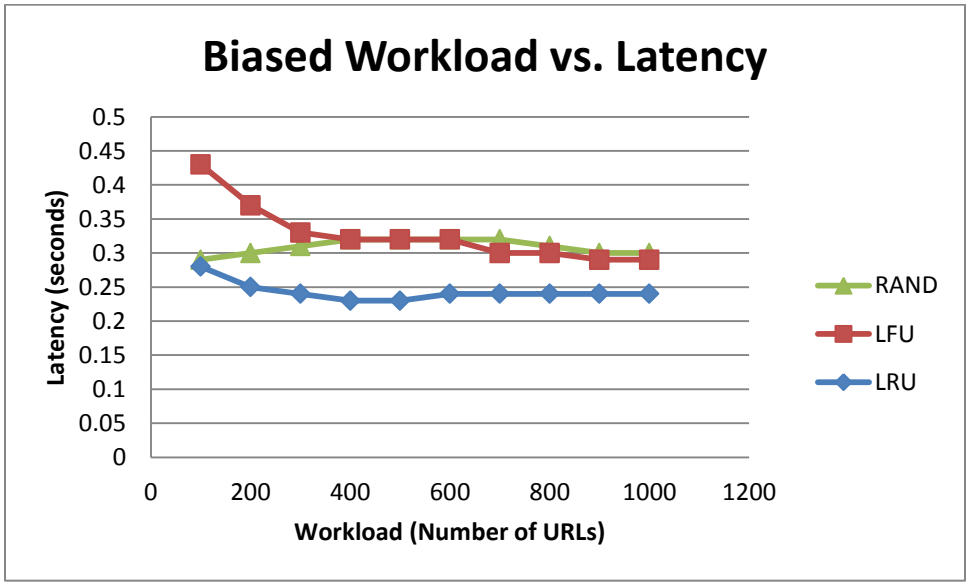
Cache Size = 3MB



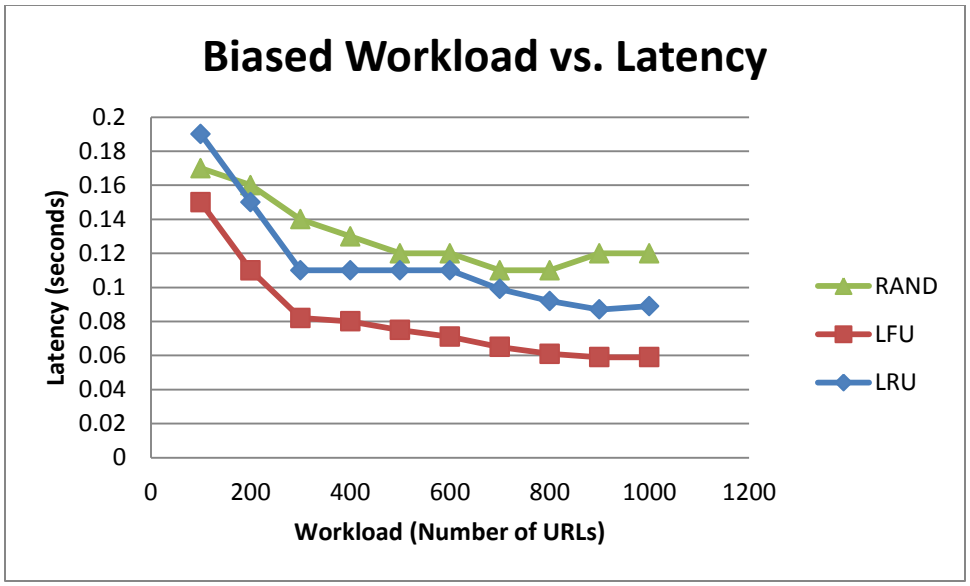
Cache Size = 5MB



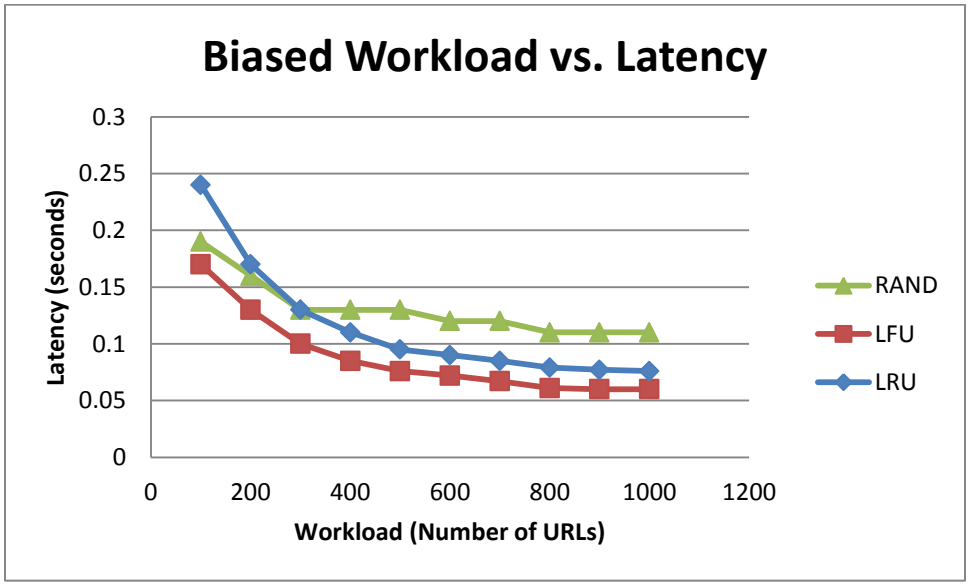
Cache Size = 1MB



Cache Size = 3MB



Cache Size = 5MB



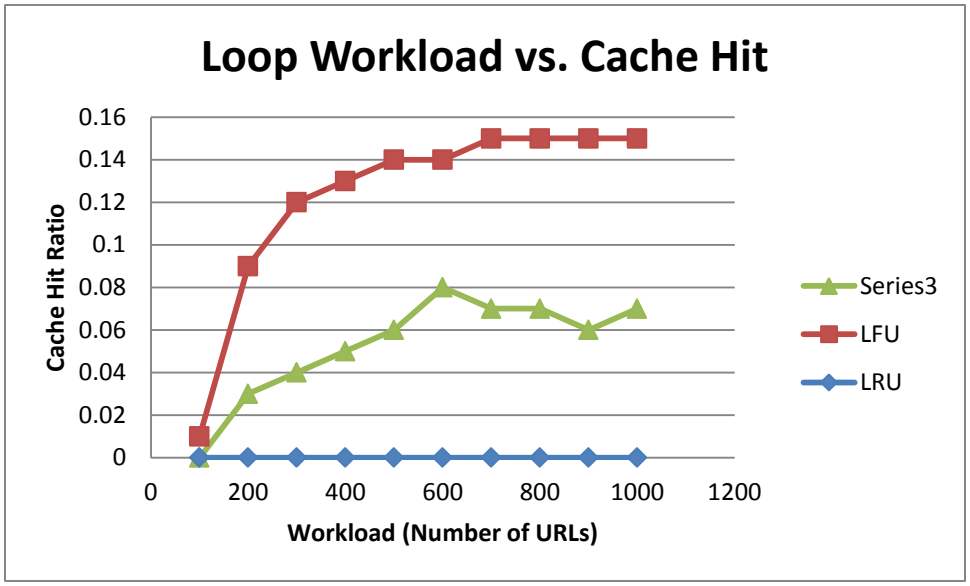
Inference:

- In theory, the biased workload favors LFU and LRU equally. But the results show that LFU performs slightly better than LRU in higher workloads because in lower cache capacities, the size of the active set of repeating patterns in the input sets of URLs just exceeds the available cache space, thereby reducing the hit ratio of LRU marginally.
- This workload pattern has little/no effect on the performance of random policy as observable in the graph.
- The latency in LFU is unexpectedly higher than LRU due to the additional overhead in maintaining the LFU order in groups of uniformly distributed cache entries of higher frequencies, which does not happen in the case of LRU.

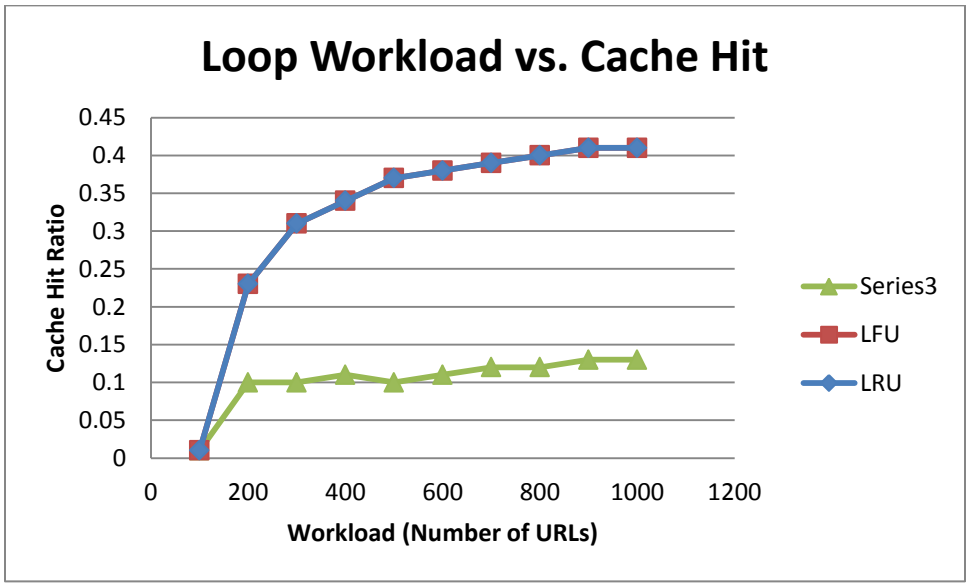
- The effect of increase with increase in cache size is observed soon with this workload (at 3MB cache size itself as opposed to 5MB in other workloads), (i.e.) the cache hit ratio stabilizes and is almost the same for LRU and LFU, but random policy is unpredictable as always.

4. contiguous redundant sets(100 in each set) of URLs just as it is in the source list

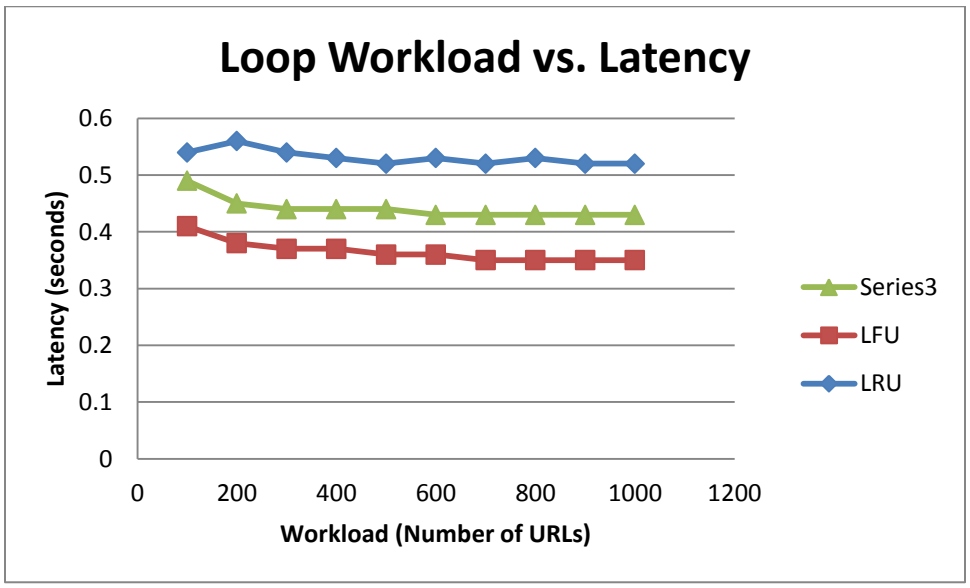
Cache Size = 3MB



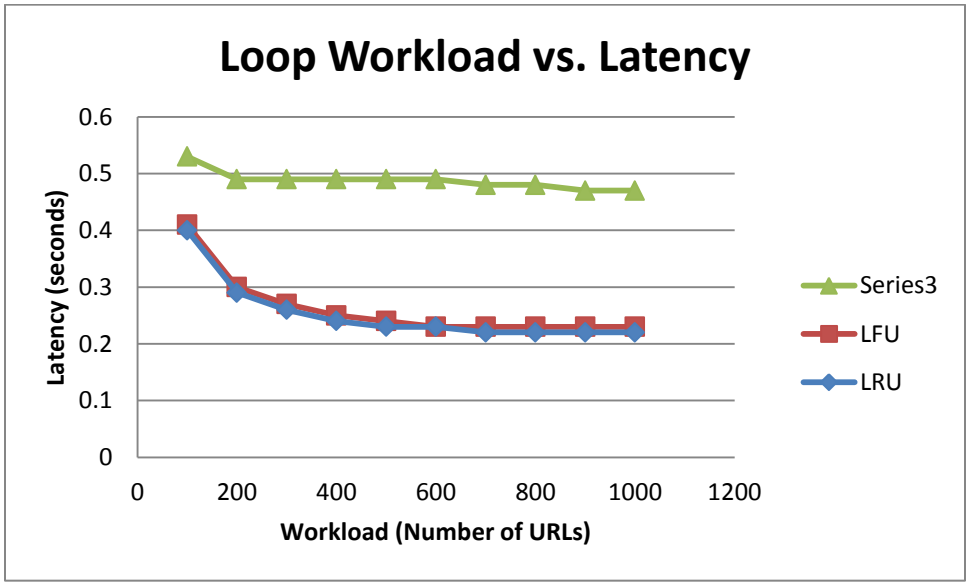
Cache Size = 5MB



Cache Size = 3MB



Cache Size = 5MB

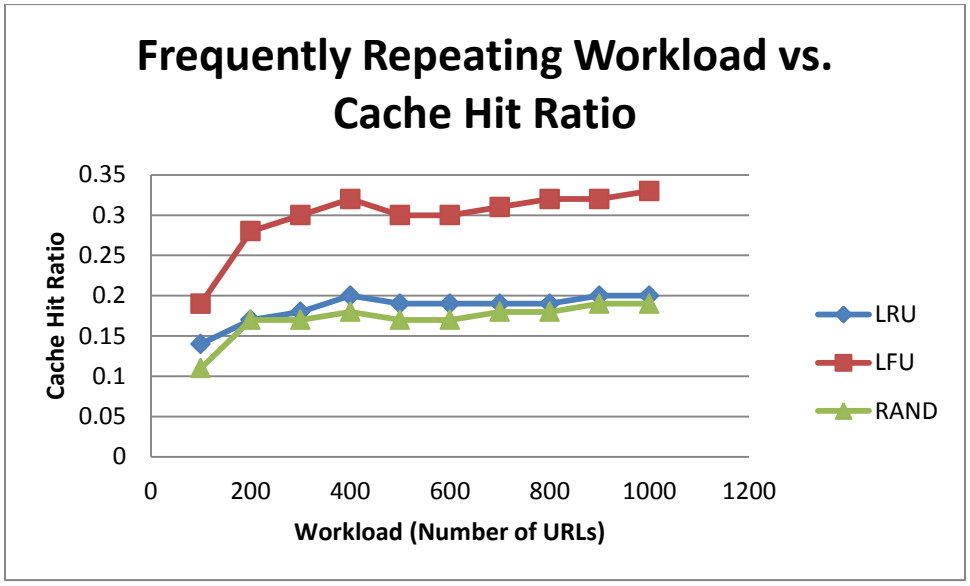


Inference:

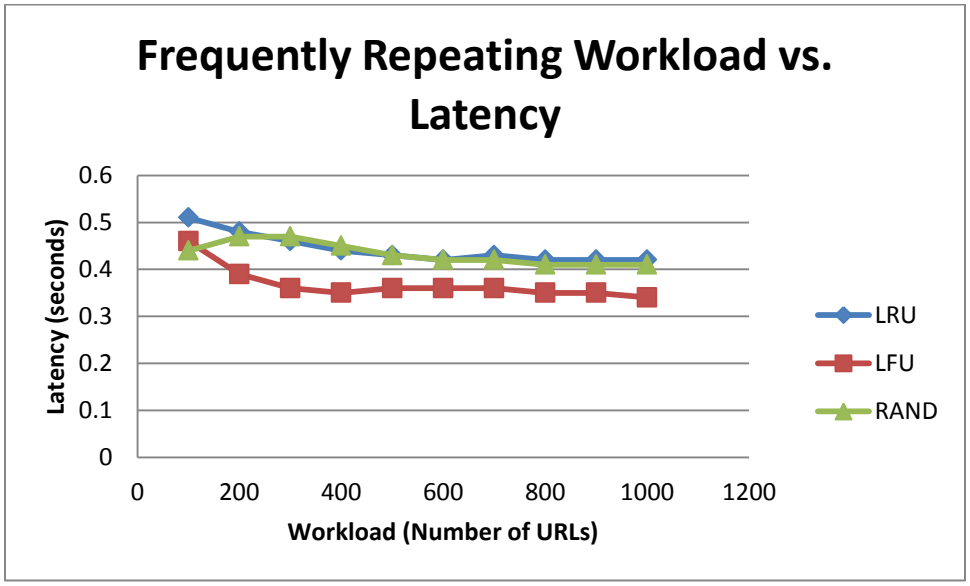
- In this workload, the minimum size of any repeating pattern is very much greater than the cache capacity and hence LRU and LFU policies behave very similarly.
- In case of small cache sizes (3MB), the LFU performs slightly better due to the fact that even lesser number of hits contributes significantly to the hit ratio computation.
- The effect of increase in cache size is similar to that of the previous workload, but LRU starts performing well gradually, whereas LFU performs well at 3MB cache size itself.

5. Frequently Repeating Pattern favoring LFU

Cache Size = 3MB



Cache Size = 3MB



Inference:

- The frequently repeating pattern in this workload type favors LFU over LRU as clearly visible from the charts.
- The cache hit ratios and latency for varying cache capacities were found to produce similar results.