Event-Driven Simulator For Network Cache System

CIS6930 – Probability for Computer Systems and Machine Learning

Prateek Behera
ECE University of Florida
UFID: 1672-3280

Anjali Baheti
CISE University of Florida
UFID: 2153-8534

***Abstract*—We constructed a network cache event-driven simulator to assess the efficacy of LRU, FIFO, and Largest-First cache implementations using our knowledge of probability distributions. New file request events are created using a Poisson distribution in each simulation, while the popularity size of files on the distant server is also generated using a Pareto distribution.**

***Index Terms*—FIFO, Least-Popular-First (LRU or Oldest First), Largest-First, Cache replacement policies, Probability distributions, Pareto, Poisson**

I. Introduction

Optimal cache replacement is a challenge since it is hard to predict what files the user will request next. There are many cache replacement policies that try to find out which files have a high chance to be used later. For a similar reason, we have developed a simulator in python to compare the effectiveness of FIFO, Least-Popular-First (LRU or Oldest First), and Largest-First cache-replacement algorithms.

We utilized an abstract simulator that simulates what occurs when files are requested from a distant server in general. With object-oriented principles, we designed a simulator that used Pareto and Poisson distributions in Python. The simulator employs these distributions Object-oriented design approaches were used to construct the simulator. To generate file sizes, popularity, and new file request event timings, our simulator employs Pareto and Poisson distributions.

II. Simulator Design

Object-oriented design was used to create our simulator. Here's a quick rundown of each class that we used in the simulation.

A. File Class

A file class stores the data into the file that uniquely identifies the file such as serial id, the size of the file (in MB). These parameters helps to assign the information to the file.

B. File List Class

File List Class stores the file in a list depending on the total number of the files and the file requests.

C. Event Classes

The event class folder has an abstract parent class "event" and a child element for each distinct event that might happen during the simulation: request new file, receive new file, queue arrival event, and queue departure (at local computer) event. These sub classes all have a constructor method and a process method for handling the event. In the main driver loop of the program, a priority queue of type event* is constructed to run events. It provides links to events that are ranked in order of priority based on when they will occur. When an event runs, it returns any events it generates, which are then put to the priority queue.

D. Request New File Event Class

This event corresponds to a new user request for a file. Here we first check the cache queue to look for the requested file. If a cached copy is present, it generates a new Receive New File Event. If a cached copy isn't present, it generates a new Queue Arrival Event.

E. Receive New File Event Class

This event represents that a file has been received by the user. When processing such an event, the following need to be done. Then the response time associated with that file is calculated and response times are recorded.

F. Queue Arrival Event Class

This event corresponds to a file arriving at the in-bound FIFO queue. Firstly, if the queue is empty, a new Queue Departure Event is generated. If the queue is not empty, the file is added at the end of the FIFO queue.

G. Queue Departure Event Class

This event represents that the access link has finished the transmission of a file and its serial ID is recorded. Firstly, it stores the new file in the cache if there is enough space. If the cache is full, enough files are popped out based on the cache replacement policy and the new file is pushed into the queue i.e., is stored in the cache queue. Then a Receive New File Event is generated. If the FIFO queue is not empty, a new

Queue Departure Event is generated for the top of the queue file.

H. Cache Classes

We developed a parent Cache class and FIFOCacheMethod, LeastPopularFirstCache, and LargestFirstCacheMethod (LRU) child classes to implement the FIFO, LRU, and Largest-First caches. Although the inner workings and architecture of these classes change, their techniques remain the same. This allows us to provide a Cache pointer to any of the three Cache implementations into the constants object, which is utilized by simulation events.

I. Configure Parameters

The following is a list of the parameters that are searched from the input file and stores the input values as it changes. Input values such as : number of requests, number of files, request rate, network bandwidth, access link bandwidth, cache check time, Pareto alpha i.e. Pareto shape, cache capacity, cache replacement policy and time limit.

J. Store Statistics

Store Statistics stores all the Response Times and the Cache hits as it comes.

III. Cache Replacement Policies

A. Least-Popular-First (LRU or Oldest First)

Least-Popular-First or Oldest First or Least Recently Used (LRU) signifies that the file that has been accessed the fewest times will be removed first. The files are added to the Cache of type OrderedDict through a doubly-linked list, and each new file is moved to the top of the list. If a file that is already in the cache is requested, it will be moved to the top of the list. If the cache fills up and a new file arrives, it will be placed to the front of the linked list, and the file at the end will be popped out of the queue.. At each file serial ID, the file size is saved to a map.

B. FIFO

The acronym FIFO stands for First In First Out, and it functions as a queue. This implies that once the cache is full, it will begin deleting the oldest file added and putting the new one to the back of the queue. To do this, we utilized a queue and map object to store the file serial ID and size, allowing us to retrieve the file information and determine which file will be erased next.

C. Largest First

LargestFirst signifies that the file that has the largest size in the cache queue will be removed first. The files are added to the Cache of type OrderedDict, and each new file is moved to the top of the list if it is found in the cache queue. If Cache is full then any new file request will lead to the Largest file being popped out of the Cache and replaced by the new requested file added to the top of the list.

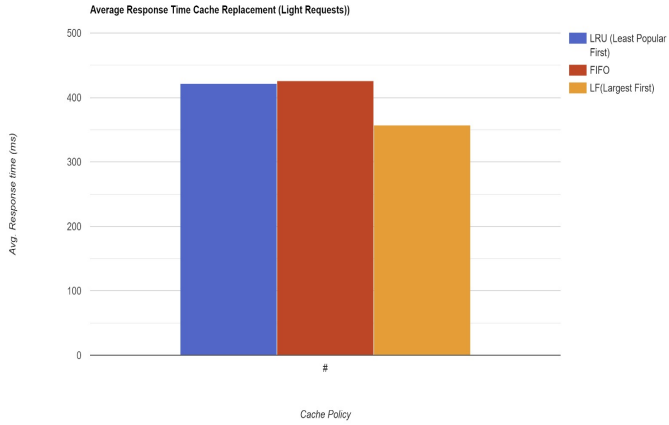
IV. Results

We examined the effectiveness of the LRU, FIFO, and Largest-First Cache replacement policies in our tests, as well as the impact of the following factors on the outcomes: cache size, number of requests, Pareto shape, Poisson mean, FIFO bandwidth, cache bandwidth.

A. LRU vs FIFO vs Largest First

In general, the largest first cache outperformed the LRU cache, which was followed by the FIFO cache. The FIFO and the LRU cache performed similarly, while the largest first cache performed considerably better. On repeated testing with different seed values, the average response times for the LRU, FIFO, and Largest-First caches were 428ms, 436ms, and 357ms, respectively.

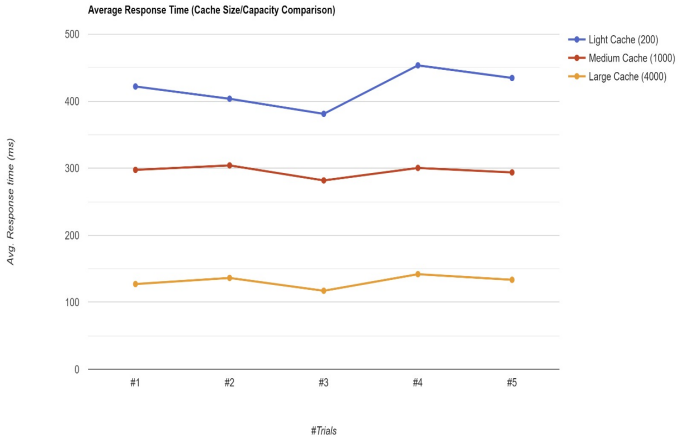
A low Pareto shape value (1.25), a medium cache size (can contain 10% of all files), and a modest FIFO bandwidth in relation to the cache bandwidth were the features that generated the largest variance between the different cache implementations. The difference in cache-performance becomes minimal at excessive levels of any input. This was especially true for high Pareto shape values; when the Pareto shape value above 2, all cache-replacement techniques performed poorly.



B. Cache Size

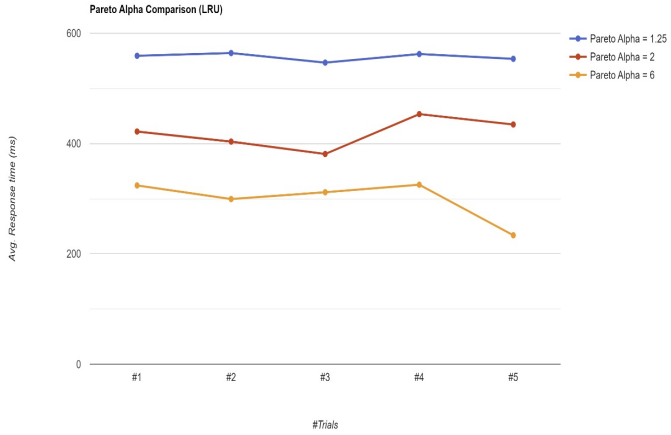
A increased cache size, as predicted, resulted in substantially higher hit rates and shorter average reaction times. The size of the cache was the most

important element in influencing average response time.



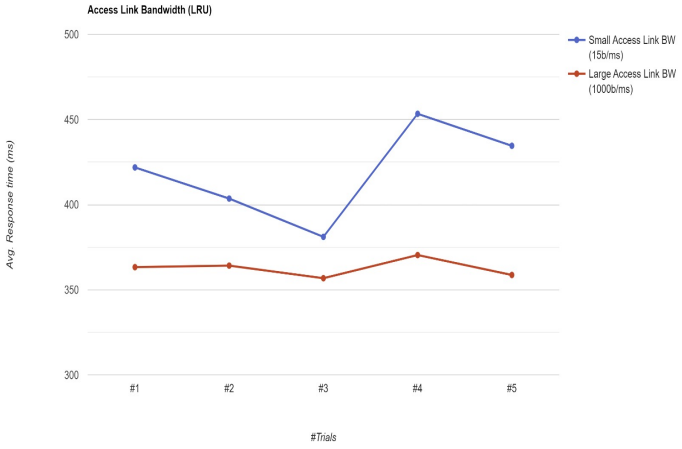
C. *Pareto Shape*

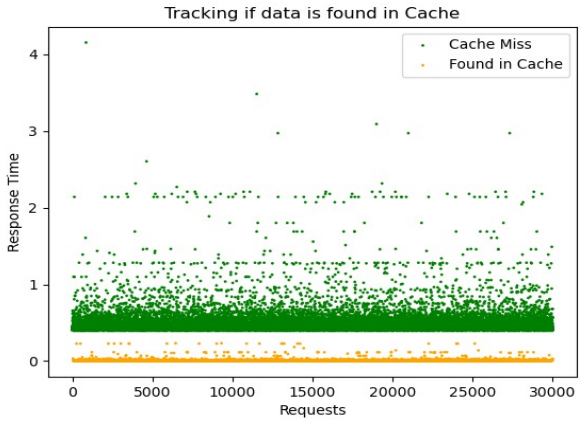
The pareto shape also had an affect on the average response time. A low Pareto shape value (around 1) indicated that the Pareto distribution will have a bigger tail, with a few very large and popular files. With a high Pareto shape value, all files will be comparable in size and popularity, making it difficult for the cache to predict which files will be requested next.



D. *Access Link Bandwidth*

Our findings reveal that when the simulation's FIFO queue bandwidth is low in comparison to the cache bandwidth, the average reaction time is substantially longer than when the FIFO queue bandwidth is large. A low FIFO bandwidth causes cache misses to take longer, increasing the time spent on a cache miss and highlighting the differences in performance between the different cache replacement policies. The second figure below shows the response times for LRU for cache hits (if file is found in cache) and misses.





V. Conclusion

There are few that can be concluded with the data provided from the results section. First, system specifications like cache size, number of requests, file size, and popularity have a far greater impact on users' average response time than the cache-replacement policy utilized in the simulator. Cache size, Pareto shape value, access link bandwidth, and the number of requests in the corresponding order were the most critical elements in determining the average response time. We discovered that the Largest first algorithm was the top performer, followed by the LRU method, and finally the FIFO algorithm. There is a significant performance difference between the cache-replacement policies when certain parameter values are used (medium cache size, low Pareto shape value, and a small FIFO bandwidth with respect to cache bandwidth). Out all the algorithms we created, the Largest First algorithm would be the best to employ for a cache. Based on these findings, the optimal strategy for boosting average file response times for network users would be to raise network system specifications (cache size, etc...). If you don't have the resources to improve those things, enhancing the cache-replacement algorithm is a low-cost solution that will yield small improvements.