Network Cache Event-Driven Simulator

Masen Beliveau, Joseph Gravelle

CIS4930 – Probability for Computer Systems and Machine Learning
Computer Science, University of Florida
Gainesville, Florida, United States of America
https://github.com/jgravellefl/Network-Cache-Event-Driven-Simulator
mbeliveau@ufl.edu
jgravelle@ufl.edu

***Abstract*— In order to apply our knowledge of probability distributions, we created a network cache event-driven simulator to compare the effectiveness of LRU, FIFO, and Second-Chance cache implementations. In each simulation, new file request events are generated from a Pareto distribution, whereas the popularity size of files in the remote server is generated from a Pareto distribution.**

***Keywords*— FIFO, LRU, Second-Chance, Cache replacement policies, probability distributions**

I. Introduction

It is impossible to know exactly which files will be requested next by a user, which makes creating optimal cache-replacement policies a challenge. Different cache-replacement policies have been developed to try and best predict which files will need to be accessed later on by a user. We developed a simulator in c++ to compare the effectiveness of FIFO, LRU, and Second-Chance cache-replacement algorithms.

We used an abstract simulator that represents generally what happens when files are requested from a remote server. We created the simulator using object-oriented design principles. Our simulator uses Pareto and Poisson distributions in order to generate file sizes, popularities, and new file request event times. These distributions were implemented using the GNU scientific library.

II. Simulator Design

Our simulator was developed using an object-oriented design. Here is a short description of each class that we utilized for the simulation.

A. Event Classes

The event class folder holds an abstract parent class "event" and a separate child element for each different event that can occur during the simulation: file request event, file enter FIFO queue event, file leave FIFO queue event, file arrive (at local computer) event. All of these subclasses contain a constructor method and a process method that processes the event. To run events, a priority queue of type event* is created in the main driver loop of the program. It contains pointers to events, which are ordered in priority based on the time that they will execute. When an event executes, it will return any events that it creates, and these will be added to the priority queue.

B. Constants Class

Pointers to resources needed by events are stored in a constants class object. A pointer to the constants is passed into each event so that it has access to the fields in constants. These fields include the cache, the remote server, the FIFO queue, a file selector object, and various variables such as the number of requests and the propagation time to access the remote server. This class was created to hold all the constants together in one place that can be accessed by each event, rather than using global variables.

C. Cache Classes

To implement the FIFO, LRU, and Second-Chance caches, we created a parent Cache class and FIFOCache, LRUCache, and SecondChanceCache child classes. Although these classes have different inner workings and structures, their methods are kept the same. This way we can pass through a Cache* pointer to any one of the three Cache implementations into the constants object used by events in a simulation.

D. Remote Server Class

In order to keep track of files outside of the cache, we used a remote server object storing (in a fileId->file mapping) all of the files generated at the beginning of the program in the form of file objects. File objects contain a file's Id and its corresponding size. File sizes were generated for each file Id using a Pareto distribution, implemented using the gnu scientific library.

E. File Selector Class

A file selector object storing a mapping of each fileId to its respective popularity (generated based on results of a Pareto distribution), is used so that each new file request event can choose a file based on the file popularity distribution.

F. Input Parameters

Here is a list of the input parameters that are entered into the program by a user: paretoMean (essentially mean file size, (in B)), paretoShape, cacheCapacity (in B), number of files, number of requests, propagation time (in ms) cacheBandwidth (in Mb/s), fifoQueue bandwidth (in Mb/s), Poisson distribution mean (basically how long on average until the next request (in ms)), cache type (the type of cached used for the specific simulation).

III. Cache Replacement Policies

As previously stated we implemented LRU, FIFO, and Second-Chance cache replacement policies for the simulation.

A. LRU

LRU stands for Least Recently Used, which means it will first remove the file that has been accessed least recently. The files enter the cache through a doubly-linked list, and when a file is added it is put to the front of the list. If a file is requested that is already inside the cache, then it will be put at the front of the list. If the cache reaches capacity and a new file comes in, it will go to the front and the file at the end of the linked list will be removed. A map is also used to save the file size at each file ID.

B. FIFO

FIFO stands for First In First Out and works as a queue. This means that after the cache fills up to the capacity, it will start removing the oldest file added and put the new one at the back of the queue. To implement this we used a deque and map object to store the file ID and file size in order to be able to access the file information and know which file is next to be removed.

C. Second Chance

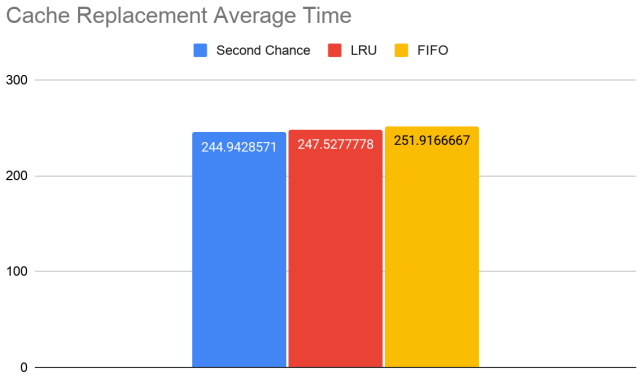
The Second chance replacement policy is very similar to FIFO with a small alteration. It still works as a queue system, but each file in the cache is also assigned an R bit which is initially set to 0. A file enters the cache by being inserted at the end of the queue. If a file is requested that's inside the queue, the R bit will be switched to 0. If the cache reaches capacity and a new file is added, it will go through the queue and check if the R bit of the file at the front is 0 or 1. If it is 1, it gets set to 0 and is added back to the queue. If it is 0, it is removed and the new file is added to the back of the queue. It keeps looping through the queue until it finds a file with R bit 0, so if all the files had their R bit set to 1, it would loop through and set them all back to 0 and remove the file at the front of the queue to be replaced by the incoming file.

IV. Results

In our testing, we compared the effectiveness of the LRU, FIFO, and Second-Chance Cache replacement policies, as well as test the effects of the following parameters on the outcomes: cache size, number of requests, Pareto shape, Poisson mean, FIFO bandwidth, cache bandwidth.

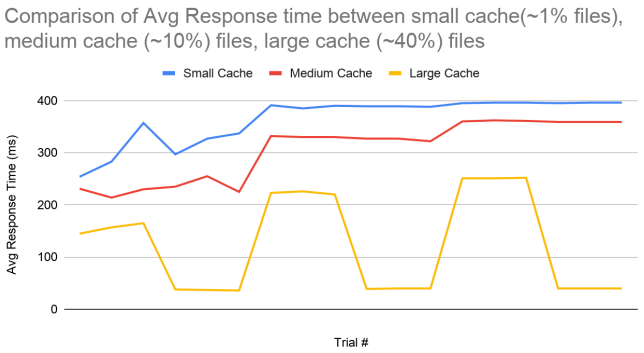
A. LRU vs FIFO vs Second Chance

In general, our second chance cache performed best, followed by the LRU cache, followed by the FIFO cache. The performance of the second chance cache and the LRU cache was very similar, but the FIFO cache was noticeably worse. Here are the average response times of the LRU, FIFO, and Second-Chance caches in our testing:: 247ms, 251ms, 244ms. The features that promoted the most difference between the different cache implementations were a low Pareto shape value (1.25), a medium cache size (can hold ~10% of all files), and a small FIFO bandwidth with respect to the cache bandwidth. At extreme values of any input, the difference between cache-performance became negligible. This was especially true for high Pareto shape values when the Pareto shape value was larger than 2, all the cache-replacement algorithms were equally bad.



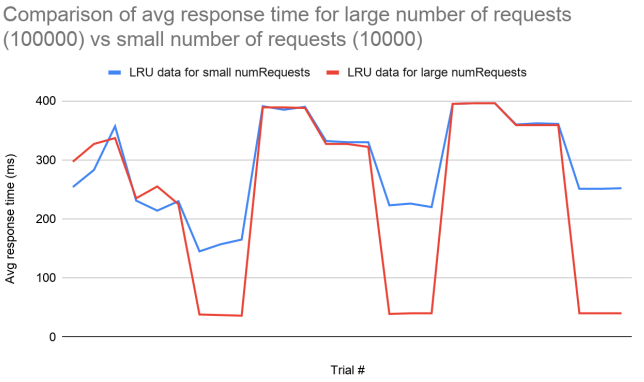
B. Cache Size

As expected, a larger cache size resulted in much higher hit rates and lower average response times. Cache size was the biggest factor in determining average response time.



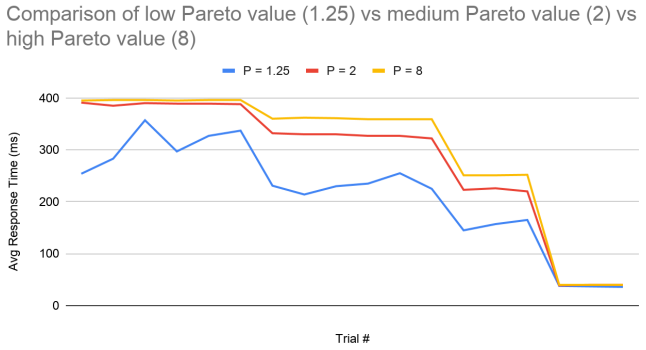
C. Number of Requests

The number of requests had a large effect on the average response time when the cache size was large but otherwise was not a huge factor. You can see clearly in the below graphic where the large cache size and the large number of requests combine (it's where there is a large gap between the lines).



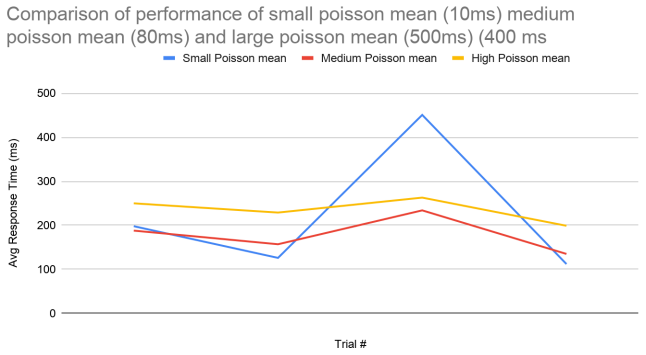
D. Pareto Shape

The Pareto shape had a large impact on the average response time. A low Pareto shape value (near 1) meant that the Pareto distribution would have a larger tail, and there would be a few really big and a few really popular files. A high Pareto shape value would result in all the files having a similar size and popularity, which would make it hard for the cache to guess which files would be requested next. Because of this, with high values of Pareto shape, the simulator does not return any interesting information, as all the cache-replacement policies are equally useless.



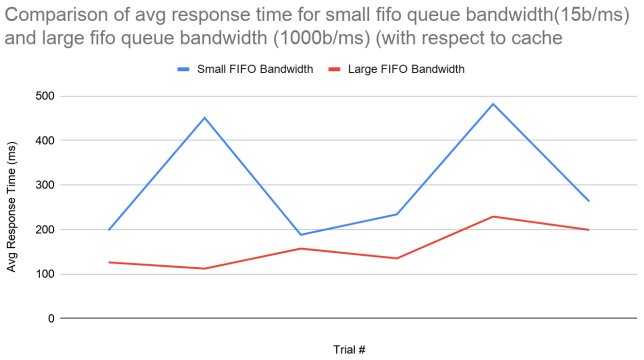
E. Poisson Mean Value

In our simulator, the Poisson mean corresponds to the mean expected value returned from the Poisson distribution. Changing the Poisson mean value gave interesting responses. It seemed to have a strong effect on the average response time under certain circumstances. This makes sense, as a high frequency of requests results in a complex situation. If you have a high frequency of different requests, it will be hard to keep a reliable cache, but if you have a high frequency of similar requests, then one passthrough of the FIFO queue will get the file that many requests need. In the below graph, the point at which the small Poisson mean results in a high average response time is when the Pareto shape is a high value, and the FIFO bandwidth is low. These two factors cause a combination of cache-misses and the low transfer time from the remote server to the cache, which is made worse by a high request frequency. We're not sure how the higher Poisson mean values result in a larger average response time in the data we collected (in the below graphic). This may be due to an error in our simulator, although we weren't able to find one in our testing.



F. FIFO and Cache Bandwidth

Our results show that when the simulation has a low FIFO queue bandwidth with respect to the cache bandwidth, the average response time is much higher than if the FIFO queue bandwidth is high relative to the cache bandwidth. A low FIFO bandwidth makes cache misses take an extra amount of time, so it increases the time taken on a cache miss, and make obvious the difference between the performance of the different cache replacement policies.



V. Conclusion

From the data discussed in the results section, a couple of things are clear. First, the system specs (cache size, number of requests, file size, and file popularity distribution) have much more of an effect on the average response time experienced by users than the cache-replacement policy used in the simulator. Certain parameter values result in a large performance difference between the cache-replacement policies (medium cache size, low Pareto shape value, and a small FIFO bandwidth with respect to cache bandwidth). The most important factors (in order) for determining the average response time were cache size, Pareto shape value, FIFO queue bandwidth with respect to cache bandwidth, and the number of requests, We found that the second-chance replacement algorithm was the best performing algorithm, the LRU algorithm was second-best, and the FIFO algorithm was the worst. Our second-chance algorithm would be the best algorithm to use for a cache out of the algorithms that we implemented. Based on these results, in order to increase average file response times for users in a network, the best solution would be to focus on increasing network system specs (cache size, etc…). If no resources are available to increase those things, then improving the cache-replacement algorithm is a cheap option that will provide modest results.