An Event Simulation of Network Activity

Jacob Murley  
*Computer Information Science and Engineering*  
*University Of Florida*Gainesville, USA  
jacobmurley@ufl.du

*Abstract*— This paper details a unique event driven simulation that models network requests using a FIFO queue and a cache. The Simulation is then used to explore replacement policies for the cache.

# Introduction

An event simulator creates a bunch of events and then processes those events. It is particularly helpful in identifying possible bottlenecks in a system. For example, in the current simulator if the files are often not in the cache then the FIFO queue can represent a significant bottleneck and can cause average time for a packet to be received to skyrocket from just .05 seconds to .97 seconds. It can also be helpful to test certain policies without having to test those policies in the real world. The policies can be tested in a variety of circumstances.

# Unique Aspects of Simulator

Most event driven Simulators use events to spawn additional events. However this simulator spawns so many events per second. The advantage of this type of simulation is that it models something that collects packet requests and them sends them out on a timer. If there was a network router that sends out all of the packet requests on an interval this way of simulating events would more accurately represent what is happening. There are some other unorthodox ways this code works. For example, at the FIFO queue with unlimited capacity you will notice that each event is inserted into the queue and immediately removed. This leads to the question on how this queue could accurately represent multiple packets waiting in the queue. The queue has a time variable and whenever an item is processed it moves the time forward by the amount of time it takes to process that packet. The time in the FIFO queue is determined by subtracting when the packet arrived the queue and when it left the queue. Thus the integrity of the time taken is maintained even with only one event in the queue at a time. If the packet arrives at the queue and the time of the queue is behind the time of the packet, then the queue was empty when the packet arrived, and the queues time is set as the time of packet arrival before processing.

Additionally, the method of choosing the files can be confusing. After assigning each file a certain probability drawn from a pareto distribution I wasn’t sure how to draw those files according to the probability that they are to be selected. I broke it up into steps. Step one was to generate the probabilities and once all of them are generated divide each by the sum of all of the probabilities. Step two was to then convert those probabilities to a whole number and assign them to the file. I did this by multiplying the probability by a factor of ten. Step three generate a map that maps the following P(i) numbers to it’s associated file in a map. Step four was drawing number from the range of the map at a uniform distribution. I did it this way because I know that the lookup time of a map in C++ is O(1) so once I created a map I could easily find the file associated with a number. I’ll illustrate what I mean with an example below.

1. Draw Number from pareto for file 4, .86 is drawn and divided by the sum of probabilities to be .0007
2. The number is multiplied by 10000 so it is now 7
3. 4 is inserted into the map 7 times
4. A number, r, is drawn from the uniform distribution covering the range of the map. If map[r] = 4 than we generate a request for file 4.

# Cache Replacement Policies

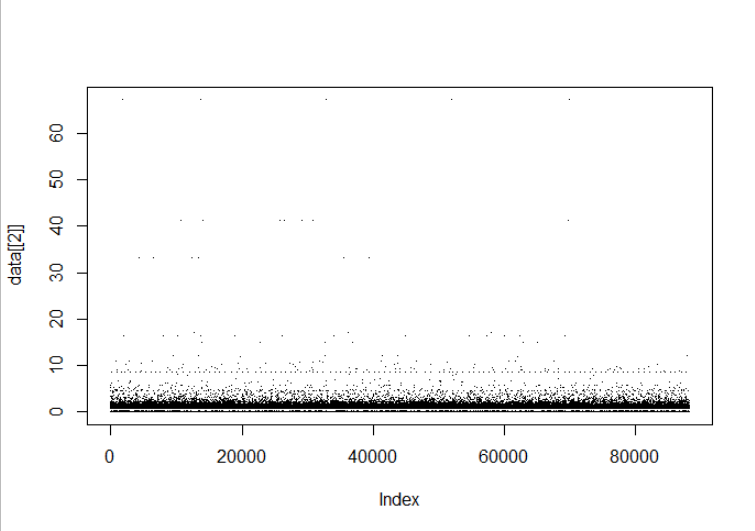
I used two cache replacement policies the first policy is first in first out so that the file that is in the cache the longest is removed. While, it intuitively makes sense to remove the oldest file, in reality this policy is often inefficient because other replacement policies like least recently used are much more effective. The faulty assumption is that the oldest file is less likely to be needed again. Oftentimes a small proportion of files are very popular and others are not. Replacing the oldest file does not reflect this behavior.

The second Replacement Policy I used was removing the largest file. My thinking behind this one is that by removing the largest file the cache can have a higher hit rate by nature of having more files. I think my assumption once again is that I’m drawing files with uniform randomness, when I know that the files are drawn according to a pareto distribution. Therefore, I don’t think this replacement policy will perform very well on a real use case.

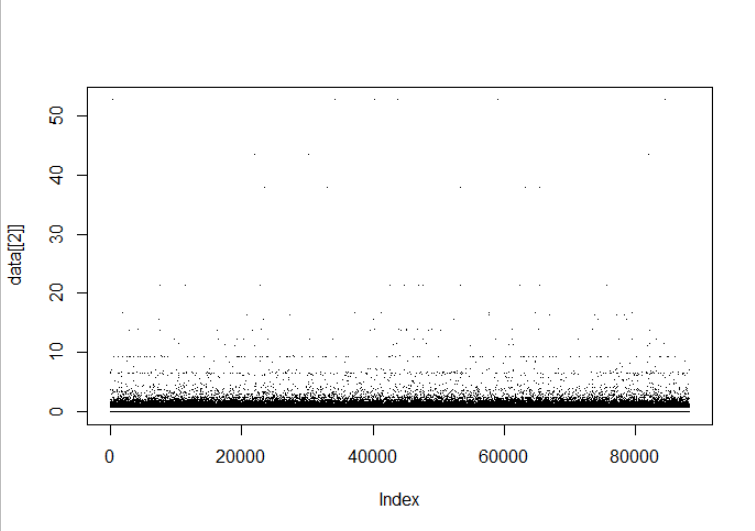
# Results

The two file replacement policies performed similarly. Below is my first run of the simulator.

Oldest



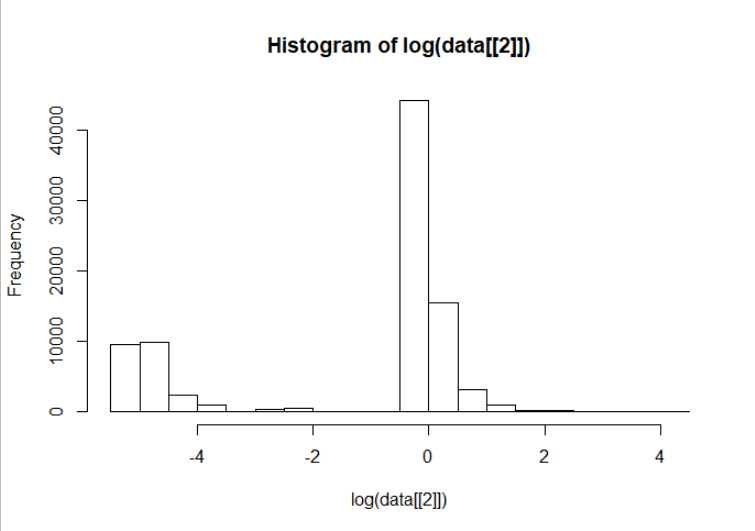
Largest



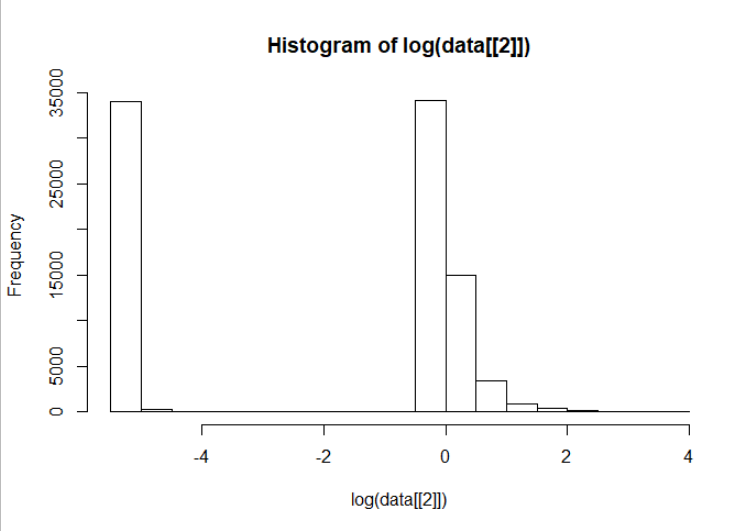
The graphs look remarkably similar however the largest had more severe cache misses more often. Nonetheless it had fewer cache misses resulting in an overall mean of 0.7034194 compared to the Oldest mean of 0.7968518. Making the Largest on average a tenth of a second faster.

The differences in hate rates is more obvious when looking at the associated histograms.

Oldest:



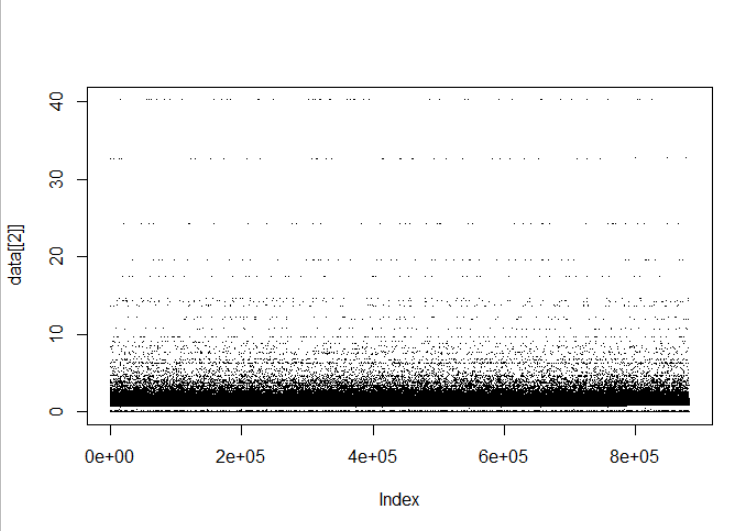
Largest:



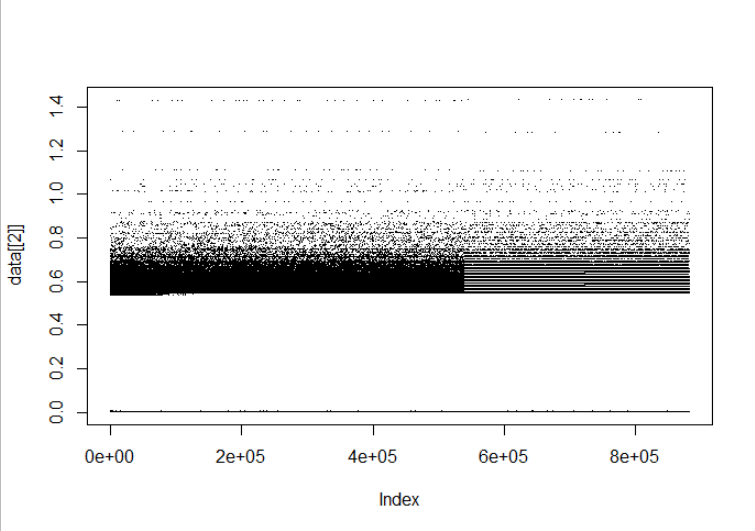
It is obvious that the oldest replacement policy had less time consuming cache misses. But the largest made up for it with the vast amount of cache hits.

After adjusting the pareto distributions to increase the popularity density of files and decrease the proportion of large files while keeping the mean file size the same the following charts resulted.

Oldest:

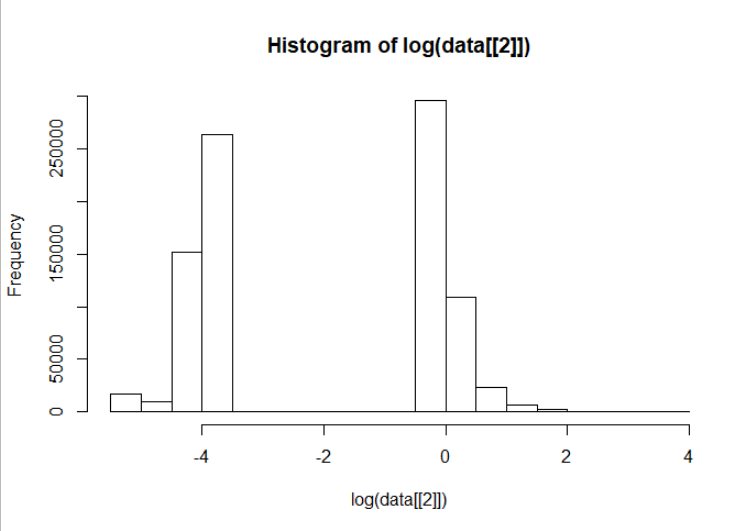


Largest:

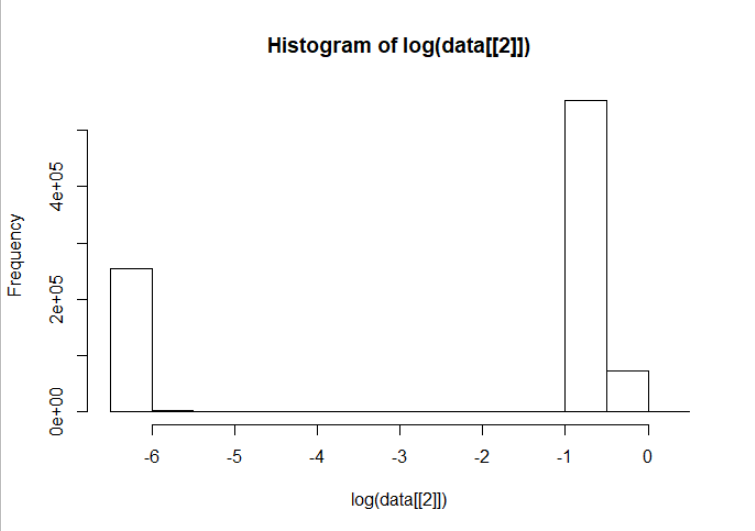


The Oldest had a less stratified distribution, while the largest eventually settled into these grooves. The mean of the oldest was .54 with a median of .02. On the other hand, the mean of the largest was .41 with a median of .55. So the two replacement policies skew in opposite directions, and once again largest is faster. Looking at the histograms,

Oldest:



Largest:



Using the histograms it seems the cache hits of the largest are able to make up for the slower cache misses.

##### Conclusion

Based off of the results of the simulations a largest file removal algorithm results in faster average file download speeds in this system. This is mostly due to the higher cache hit rates that are generated by having more files in the cache.

More research is needed on other algorithms such as Least Recently Used before a final recommendation can be made for this system.