**Simulating and Measuring Hit Rates for Various Cache Replacement Policies**

Sean Mebust and Jack Fergus

**Introduction**

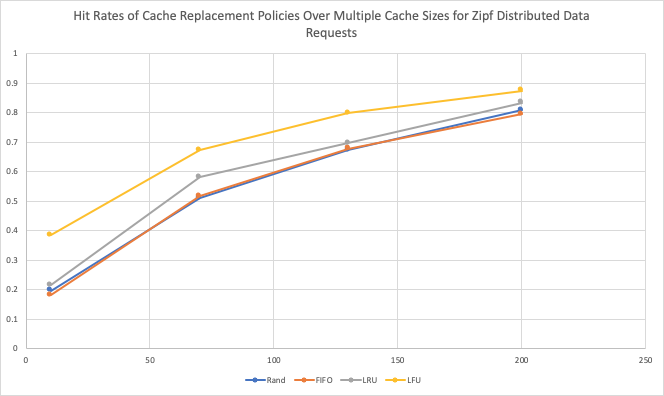
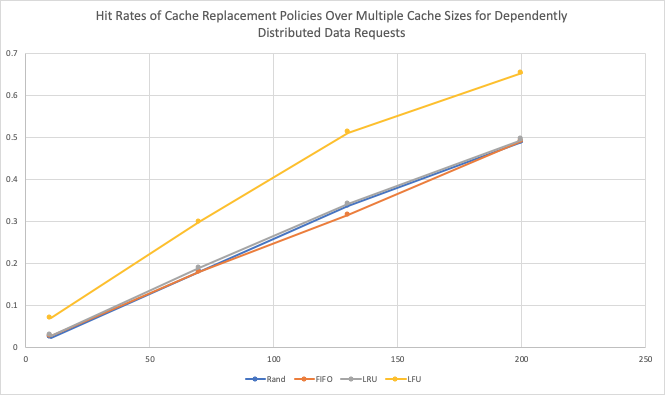
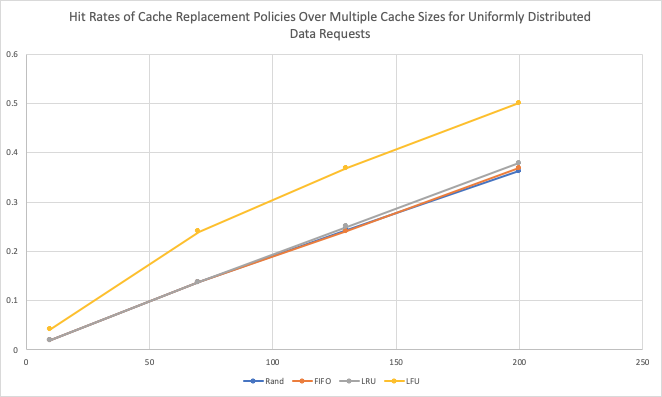
The goal of this study was to observe the effects of different cache replacement policies on the hit rates of cache requests. Caching is an important and useful procedure in computer systems that speeds up the process of requesting data from memory. A request for a piece of data stored in RAM, for example, could take fractions of a second to complete. This seems like a small amount of time, but in the total operation of the system, repeated requests have the potential to seriously slow down performance, since the program must wait on every piece of information requested from memory. Caching is a way to avoid this issue by storing a small amount of information in a much more quickly accessible place. A cache hit is when a data request is made, and the program is able to find the data stored in the cache rather than having to search for it in RAM or some other, slower access memory space. It is important to have a cache replacement/cache management policy with a high rate of cache hits, because this will drastically increase the running speed of programs using the cache. A higher hit rate means that a high number of data requests are complete through accessing the cache, which results in much faster data request completion. The goal of this project was to simulate different policies for cache management in order to see how different methods for replacing cache items affects the hit rates for cache requests. In order to maximize the cache hit rate, we had to see how each policy performed on multiple data request distributions.

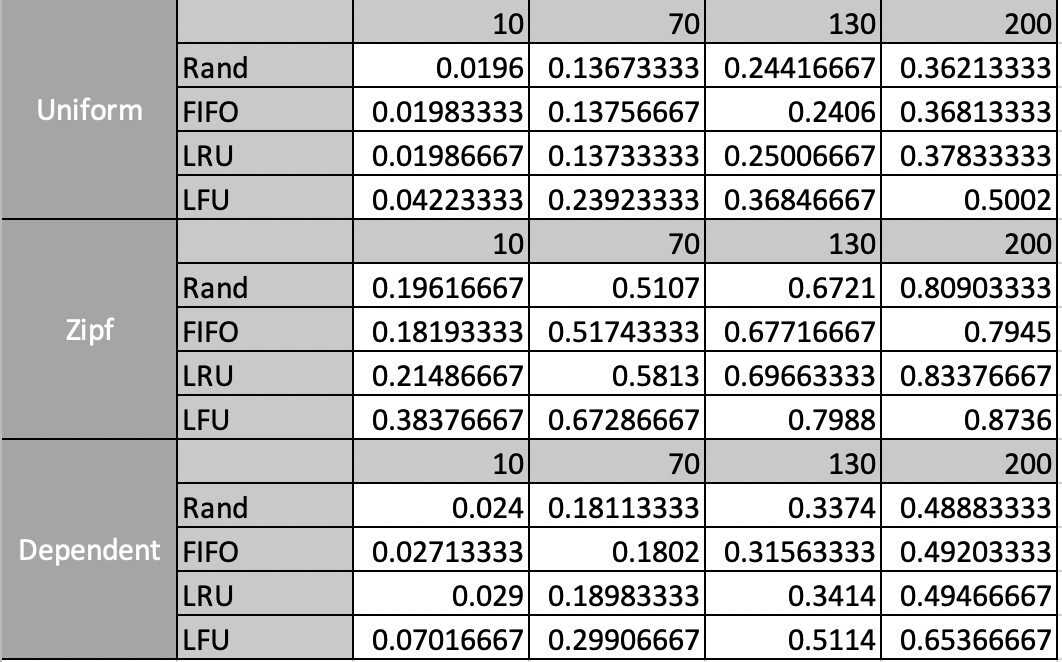
**Experimental Setup**

Our simulator takes in user-input parameters for distribution type, cache size, and cache replacement policy. We ran experiments by running the program three times each on 48 different combinations of parameter settings, and averaging the results of these three trials for an output of 48 simulated hit rates. Our data requests were drawn from a RAM array of size 1000, which contained values between 1 and 1000 distributed differently depending on user specifications.

We first ran experiments using a uniform distribution of data requests. Selecting a uniform distribution results in the array being filled with data uniformly with values between 1 and 1000, so each value has an equal likelihood (1/1000) of being chosen. We then looked at outputs from requests drawn from a zipf distribution. This distribution causes the array to be filled with zipf distributed values between 1 and 1000. In this case, each successive value is significantly less likely to be requested than the previous value. As the value increases, its likelihood of being requested approaches 0. This results in the same low values being requested many times over, and higher values rarely being selected. Our third and final distribution of data requests was one in which each successive data request is not independent from the previous request. Selecting this distribution results in the RAM array being filled with 1000 dependent values generated by a provided RequestGenerator class. All three of these methods created a distribution by filling an array of size 1000 with values drawn from a specified distribution, which was then used for data requests to the cache.

Our simulator utilized a cache size specified by user input. When performing our experiments, in order to avoid running thousands of trials, we chose four different cache sizes to test: 10, 70, 130, and 200. We tested each of our replacement policies on each of our request distributions four times, once for each of these four cache sizes. We tested four different replacement policies which, when running the simulator, were selected based on user input. Our Rand() method performed random cache replacement in the case of a miss, meaning a random value in the cache was selected and replaced with the requested value. FIFO() performs first in first out cache replacement, for which we utilized a queue to represent the cache. In the case of a cache miss, this method removes a value from the end of the cache and inserts the requested value at the beginning. LRU() performs least recently used cache replacement. In the case of a miss, this method selects the cache slot containing the value that, when compared to all other values stored in the cache, has been requested least recently, or furthest in the past. Similarly, our last method, LFU(), performs cache replacement by selecting the cache slot containing the value that, when compared to all other cache values, has been requested the least so far.



****

**Results and Discussion**

We made three predictions before running our experiment. First, LFU would perform the best (have highest hit rate) out of all the replacement policies for all of the distributions. Second, we predicted that using Zipf distributed values would result in higher hit rates for all replacement policies. Third, that Random replacement would have the lowest hit rates for all of the distributions. Our first two predictions were correct, but our third prediction took an interesting turn.

As expected, LFU had much higher hit rates than the other replacement policies for all distributions, and had especially high hit rates for Zipf and Dependent distributions (.87 and .65 at cache size 200, respectively). Furthermore, the average hit rate across replacement policies for Zipf distributions is much higher than the hit rates for Uniform and Dependent distributions, for all cache sizes. What we did not expect was that FIFO and LRU would perform the same as Random replacement on all distributions. In fact, with the exception of the LRU spike for the Zipf distribution at cache size of 70, the hit rates for LRU, FIFO and Random are all within .05 of each other for every distribution at every cache size. When we ran the experiment for dependently distributed values at cache size 130, we were shocked to see that Random actually outperformed FIFO; picking a random value to replace was more accurate than replacing the first item to be placed in the cache. While this may just be an anomaly, there could be a very good reason behind it.

That reason is temporal locality. The same requests are repeated over a short span of time, but far enough apart that when using FIFO, the cache will no longer contain the requested value. This results in having more misses than just replacing a random item in the cache. This leads us to believe that the RequestGenerator class will produce this sort of pattern, repeating the same values after roughly 1000 requests.

**Conclusion**

Our results showed that for all distributions, LFU had the highest hit rates out of the replacement policies, while FIFO, LRU, and Random had roughly equivalent hit rates. Moreover, the Zipf values had much higher hit rates than Uniform or Dependent values. There were two major assumptions we made when running these experiments. We assumed that the RAM had 1000 addresses, and we assumed that all of our requests were independent. In practice, the size of the RAM differs. For our experiment, if we wanted to increase the size of the RAM, we would also need to increase the range of values that we get from the distributions, or else more and more distinct values will appear in the cache and the hit rates will increase. The second major assumption was that all of our requests were independent. This assumptions tends not to hold in practice. A lot of sequences exhibit temporal locality, where the same requests are repeated in a short amount of time. If we were to repeat this experiment and accounted for temporal locality, I think the only thing that would change is that LRU, FIFO, and Random would be even more similar. In conclusion, our experiment leads us to believe that, subject to these assumptions, LFU is the optimal replacement policy, and that Zipf distributed values have the highest hit rates.