Evaluation of Cache Replacement Policies Through Event-Driven Simulation

Pablo Hernandez-Perretti
Computer Science
University of Florida
Gainesville, Florida
pablo.hernandezp@ufl.edu

Sava Glisic
Computer Science
University of Florida
Gainesville, Florida
savaglisic@ufl.edu

***Abstract*—The following outlines the creation and implementation of event-driven simulator for a network cache system, as well as findings from this implementation.**

Keywords—simulator, event-driven, python, distribution, network, cache

I. Introduction

The modern internet is built upon an elaborate system of clients, servers, and the complex networks connecting each client and server to one another respectively. As the information age has progressed, both the files to be accessed and the number of requests for files grew larger, leading to a need for superior systems of data management. Although well designed, the networks that form the internet must be augmented by software to properly handle varying loads of user requests. In many implementations, this is accomplished with the use of a local cache. When a client attempts to access a file from an external server, this file is downloaded to the client's local network, and may be added to a local cache. The local cache allows for quicker future access to the previously accessed web data than would otherwise be possible. This improves load times for all clients on a given institutional network and reduces strain on often limited external bandwidth. However, the modern internet is extremely vast, and it is not possible to permanently cache all data accessed from external servers. Consequently, most modern cache implementations utilize one of a variety of cache replacement policies. These replacement policies may include a "FIFO", or first-in-first-out policy in which the oldest cached item is the first item to be removed and subsequently replaced by a new item once the cache storage constraints are met. Conversely, "LIFO" or last-in-first-out policies replace the newest item in the cache with subsequent requests once storage constraints are met. Although reliable, these cache replacement policies are less than ideal in terms of overall efficiency. In order to combat this lack of efficiency, more advanced cache replacement policies can be implemented. For instance, items in the cache can be assigned a popularity value based upon the number requests per item on the local network, this popularity cache will eventually only contain the most accessed files, providing a high efficiency means of storing data. Similarly, items in a cache can be ranked according to size, allowing for the automatic removal of very large items that consume large amounts of space on the cache. One can reasonably measure the success of a cache implementation based upon the length of time it takes for a client request to be fulfilled by the network, regardless of whether the target file originates in the cache or on an external server. Many factors including the size of cache, institutional network bandwidth, in bound bandwidth, and the propagation time of a file have an impact on the amount of time it takes for a request to be filled. In order to better understand how this variety of factors impacts the client user experience it is possible to create an event-driven simulation to effectively simulate each step of a file's movement throughout a network and cache system.

Identify applicable funding agency here. If none, delete this text box.

II. Simulator Implementation

A. Platform and External Dependencies

Due to its relative ease of use and popularity as a scripting language, Python 3 was chosen for this implementation of an event-driven network cache simulator. Standard python libraries "queue" and "random" were utilized in addition to the external library "NumPy". The "queue" library includes standard python tools for implementing the LIFO and FIFO data structures needed to simulate those respective cache replacement policies. The "random" standard library allows for simple random number generation. The external tool "NumPy" is necessary for the quick and easy generation of more complicated random values employing Pareto and log-normal statistical distributions. These are necessary for simulating a realistic variance in values like file size, which cannot be a constant in a realistic simulation.

B. Code Structure

This event-driven simulator was written with standard object-oriented programming conventions. It begins with an "Event" class storing relevant attributes of individual events occurring in the simulator. This is followed by four cache replacement

policy caches: FIFO, LIFO, Popularity, and File size. This class-based structure for cache replacement policies allows the user to quickly toggle between preferred methods of caching for the simulation. Following this, a class containing there is a class containing the event-simulator itself. The simulator accepts 11 total user defined parameters: Mu (The mean file size), N (The global number of internet files), λ (The number of requests per minute), Simulation time (the length of simulated time the simulator is executing in seconds), Cache Policy (An integer 0,1,2, or 3 representing each of the cache replacement policies respectively), Cache Size (The size of the cache in MB), Rc (The institutional network bandwidth), Mean D (The mean propagation time between the institution and external server), Standard Deviation of D (The standard deviation of propagation time), Ra (The in-bound bandwidth limitation), and Alpha (The shape parameter of the Pareto distribution). The end of the script provides a clear field for modifying this set of parameters. Once running, the simulator outputs average response time and cache hit rate (The proportion of accessed files being present on the cache).

III. Cache replacement policies

This program implements four primary cache replacement policies, each can be toggled as a numerical value in one of the 11 total user defined parameters.

A. FIFO Replacement

FIFO, or first-in-first-out cache replacement is largely equivalent to accumulating a certain mass of files such that the empty space in the cache is depleted. Once space is depleted, empty space is created by removing the "first-in" or oldest file in the cache. Such an implementation is ideal under the assumption that older files are likely less relevant to current users than the newer files currently being accessed. For example, a user accessing weather data on a daily basis is less interested in weather from two weeks ago than they would be interested in today's weather. A FIFO cache accommodates this theoretical use case well.

B. LIFO Replacement

LIFO, or last-in-first-out cache replacement is equivalent to accumulating a certain mass of files, such that the empty space in the cache is depleted. Once space is depleted, new space is created by removing the "last-in" or newest item in the cache. Such a policy leaves a cached backlog of older files that can be quickly accessed. This may be useful in a case where older files are large, important, and need to be quickly but infrequently accessible.

C. Popularity Replacement

Rather intelligently, a popularity-based cache replacement policy relies upon a set of data corresponding to each file that is able to numerically quantify its popularity. This could theoretically be accomplished by enumerating the quantity of access requests for a given file and incrementing a corresponding popularity value for such a file. Most popular files are sorted into the cache, with the least popular file being removed as soon as space is depleted, the next most recently accessed file replacing it.

D. File Size Replacement

Another intelligent cache replacement policy involves analyzing the file size that corresponds to each accessed file and sorting the cache in such a manner that the largest file sizes are automatically removed and replaced by new, smaller files. This is done to ensure that large files that may or may not be accessed don't take up an unreasonable proportion of available cache space. This may be particularly useful in a situation where many small files are frequently accessed, and a small number of large files only occasionally accessed.

IV. Findings

For Tables I-IV load varies upon λ (Requests per minute)

A. Size-Based Cache Performance

It is apparent that in utilizing the file size replacement policy with a consistently increasing load: light, medium, or heavy, Response times increase, and the cache hit rate decreases. As a whole, it is clear that response time and hit rate are directly correlated. Lower cache access rates are consistently resulting in poorer response times and thus an inferior user experience. With increasing load, cache hit rate declines from 55% to 40%, a 15% reduction in efficacy (See Table I).

B. Popularity-Based Cache Performance

In utilizing and testing the Popularity-Based cache, results were less linear. While varying between a light, medium, or high load, performance did not decline as expected with a higher load. A light load resulted in the worst response time at 1.671 seconds, a heavy load was moderately better at 1.475 seconds and a medium load performed most adequately at 1.001 seconds. Hit rate was also significantly more successful under medium load at 62.1% cache efficacy (See Table II).

C. Last-In-First-Out Cache Performance

In testing more simplistic cache replacement policies results continued their nonlinear trend. The program's LIFO cache was marginally more successful at medium or heavy load vs. light, which performed least effectively. Medium and Heavy loads performed nearly identically with 64% and 63% hit rates respectively (See Table III).

D. First-In-First-Out Cache Performance

FIFO cache replacement seems to further disrupt the trend established by more advanced policies. It performs best under heavy load, with a response time of just 1.284 seconds, compared to a 1.541 second response time under medium load. Under heavy load, the FIFO cache also managed to achieve a nearly 8% higher than medium, and about 1% better than the light load hit rate on average (See Table IV).

TABLE I.

Size-Based Cache Performance		
Scenario	***Response Time***	***Hit Rate***
Light Load	1.160	0.553
Medium Load	1.306	0.428
Heavy Load	1.412	0.407

[a.] Light, Medium, and Heavy Loads correspond to varying new file request frequencies.

TABLE II.

Popularity-Based Cache Performance		
Scenario	***Response Time***	***Hit Rate***
Light Load	1.671	0.328
Medium Load	1.001	0.621
Heavy Load	1.475	0.475

[b.] Light, Medium, and Heavy Loads correspond to varying new file request frequencies

TABLE III.

Last In First Out Cache Performance		
Scenario	***Response Time***	***Hit Rate***
Light Load	1.100	0.521
Medium Load	0.946	0.643
Heavy Load	0.996	0.631

[c.] Light, Medium, and Heavy Loads correspond to varying new file request frequencies

TABLE IV.

First In First Out Cache Performance		
Scenario	***Response Time***	***Hit Rate***
Light Load	1.488	0.475
Medium Load	1.541	0.406
Heavy Load	1.284	0.487

[d.] Light, Medium, and Heavy Loads correspond to varying new file request frequencies.

Fig. 1. Example of a figure caption. (*figure caption*)

Figure Labels: Use 8 point Times New Roman for Figure labels. Use words rather than symbols or abbreviations when writing Figure axis labels to avoid confusing the reader. As an example, write the quantity "Magnetization", or "Magnetization, M", not just "M". If including units in the label, present them within parentheses. Do not label axes only with units. In the example, write "Magnetization (A/m)" or "Magnetization {A[m(1)]}", not just "A/m". Do not label axes with a ratio of quantities and units. For example, write "Temperature (K)", not "Temperature/K".

CONCLUSION

In running repeated trials that span the simulation's parameter space, we have gleaned useful insight into both the efficacy of various cache replacement policies as well as the strength of the influence on performance results held by biases in the statistical processes which underly the simulation.

Although we have observed several instances throughout our data which indicate a non-negligible positive influence of decreasingly naïve cache replacement policies on file request response times, averaging, performance across our many samples confirms that altering the behavior of the stochastic processes via methods such as mean or alpha manipulation outweighs the benefits of improved cache replacement policies.

Moreover, we have observed an interesting and unsurprising correlation between cache size and cache hit rate, regardless of the parameter space or replacement policy. However, a more interesting and insightful notion displayed in the graphic below is that Small Cache trials refer to caches of size 50MB, whereas Large Cache trials represent those with cache sizes of 100MB.

Accordingly, it is important to note that tripling the size of the utilized cache had relatively small impact on the cache hit rate. This has led us to conclude that the optimal strategy for improving cache performance is to create intelligent cache replacement policies which better predict user behavior, rather than scaling cache storage to increase the baseline probability of a cache hit.