Implementing and Evaluating a Hyperbolic Cache

Reuben Agogoe, Stephen Dong, Jimmy Hoang

Project Description

For our final project, we implemented and evaluated, in Go, a hyperbolic cache, which is a cache that builds off of the LFU caching algorithm. While hyperbolic caching does consider the frequency at which items are accessed, it also includes a per-item notion of time, allowing the cache to avoid the inflexibility of having additional data structures to maintain a total ordering of the cached items. This unique modification seeks to combat the problem of LFU caches not being able to consider the popularity of an item over time.

The process of eviction within a hyperbolic cache begins with random sampling. After drawing a random sample of S items in the cache, the algorithm then evicts the sample item that yields the lowest priority from the hyperbolic priority function that is defined as the following:

$$p_i = \frac{n_i}{t_i}$$

where p_i is item i's priority, n_i is its access count while in the cache, and t_i is the total time since it first entered the cache. As opposed to traditional LFU caching that overly punishes new items, hyperbolic caching allows for new items to converge to their true popularities from initially high estimates. By combining random sampling with the hyperbolic priority function, hyperbolic caching is able to decay item priorities for eviction at variable rates and continually reorders many items at once, eliminating the need for extra data structures [1].

In order to evaluate our hyperbolic cache, we compared its performance, particularly hit ratio, to that of FIFO, LRU, and LFU caches that we also implemented or modified from previous assignments.

Design Overview

We designed all of our caches to fit the Cache interface below, using maps and/or linked lists for efficient operations.

Get(key string) (success bool)	Returns true if an item with the given key was found in the cache, false otherwise.
Set(operation_timestamp int, key string) (success bool)	Adds or updates an item with the given key in the cache and returns true if a successful update was made, false otherwise.
Stats() (*Stats)	Returns a pointer to a Stats struct that contains how many hits and misses this cache has resolved over its lifetime.

Because hyperbolic caching involves taking samples of a given size S for evictions, we defined the maximum capacities of our caches to be the maximum number of storable items as opposed to a maximum byte size limit. This way, we could simplify each item to implicitly be the same size so that we could be guaranteed at least S items in the cache before taking a sample for hyperbolic caching evictions. As a result, we did not need to store any actual (potentially large and expensive) values in memory for each item to compute hit ratios.

Focusing on our **HyperbolicCache** struct, we designed it to cache **HyperbolicCacheItem** structs that contain metadata about their access counts and initial insertion times into the cache. Using the metadata inside these items, we were able to devise a way to efficiently find the item in a sample with the lowest priority through the functions below.

calc_P() (priority float32)	Returns the priority of an item using the hyperbolic priority function.
evict_Which() (key string)	Returns the key of the item to evict based on lowest priority in a sample.

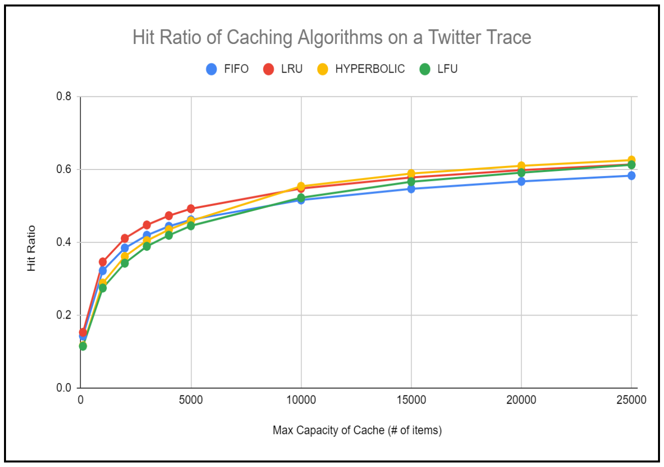
Testing

1. Correctness Testing: We wrote unit tests to verify the correctness of our different caches.

2. Performance Testing as measured by hit ratio: We computed and compared the hit ratios of our different caches across different max capacities using a Twitter cache trace [2, 3].

Results

Through performance testing, we were able to identify some interesting trends in the hit ratios of the different caching algorithms. While the hyperbolic cache was initially weaker than both the FIFO and LRU caches, it eventually, at approximately 10,000 max capacity, surpassed them both. Additionally, the hyperbolic cache continuously outperformed the LFU cache across all max capacities, indicating that it is, indeed, an improved modification of the LFU cache. We concluded from our testing and evaluations that hyperbolic caching is the best scaling algorithm out of all the ones we tested in terms of maximizing hit ratio. This aligns with some of the discoveries made in [1], so although this project was challenging, it was successful and enlightening.



References

[1] Aaron Blankstein, Siddhartha Sen, and Michael J. Freedman. Hyperbolic caching: Flexible caching for web applications. *Usenix.org*. Retrieved December 12, 2022 from https://www.usenix.org/system/files/conference/atc17/atc17-blankstein.pdf

[2] Juncheng Yang, Yao Yue, and Twitter K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at Twitter. *Usenix.org*. Retrieved December 12, 2022 from https://www.usenix.org/system/files/osdi20-yang.pdf

[3] cache-trace: A collection of Twitter’s anonymized production cache traces. Retrieved December 12, 2022 from https://github.com/twitter/cache-trace