Computer Networks And Open Systems Project (Final)

1st Ahmet Burak KOÇ
Dept. Computer Eng.
Hacettepe University
Ankara, Turkey
koc.ab@hotmail.com

***Abstract*— In this project, caching algorithm is to be designed for our users to improve user experience by reducing delay and also to save bandwidth for our organization.**

Keywords— caching, computer networks

I. INTRODUCTION

In this project, caching algorithm is to be designed for our users to improve user experience by reducing delay and also to save bandwidth for our organization. In this scenario, there are 100 files with varying popularity and sizes. Users request these files based on their popularity.

We have a limited cache space where we can store some of the files after they have been requested by a user. Hence, we can satisfy some of the user requests from the cache and save bandwidth of our organization. Our aim is to maximize the amount of data that is satisfied from the cache, hence minimize the consumed bandwidth which refer is referred to as the hit_penalty in the code.

There might also be a cost associated with replacing files in the cache. As writing and removing files from the disk might be costly, we might also want to reduce the amount data replaced in the cache. This parameter is called as replacement_penalty which we can assume 0 for parts 1 and 2.

The popularity of the files are distributed geometrically where the most popular files are much more popular than the least popular files, hence, we will find that some files are very frequently requested. The sizes of the files are also geometrically distributed. Hence, some files may fill up our cache very quickly. There is no correlation between the size and popularity of a file.

Part 1: In this part, a simpler version of the problem is given. Our caching algorithm has access to information about file popularity and size. The goal is to find an algorithm which will minimize the hit_penalty.

Part 2: In this part, we will not have access to file popularity and file size information beforehand. If we need that information, we have to infer that information keeping track of user requests. This is a much challenging problem.

Part 3: In this part, we will increase the cost of replacement to 0.1. This will mean you need to keep the cache more stable not to increase the replacement cost.

II. PART 1

A. Caching Approach

In part1 algorithm, I tried to prioritize files considering their possibility to be requested (popularity), impact to hit penalty (file_size) and memory usage in cache (file_size).

The possibility of resulting 1 hit_penalty per cache usage is calculated as,

$$\frac{popularity(fileID) * hit\_penalty(fileID)}{memory\_usage(fileID)}$$

$$= \frac{popularity(fileID) * file\_size(fileID)}{file\_size(fileID)}$$

$$= popularity(fileID)$$

So files are prioritized to be cached or not with respect to their popularity. Then from the highest popularity to the least popularity, file sizes are summed and fileID is selected to be cached as long as there is enough space in cache. Selected fileIDs are written on array "cache_files" which contains fileIDs to be cached.

B. Optimisation

In optimization of the algorithm, I tried to avoid unnecessary loops. In order to do that, main loop selecting files to be cached starts if file is not stored yet and if there is enough space in cache.

C. Code

Code is written in Python and attached to this report.

D. Results

Caching algorithm is tested for 100000 file requests which have geometrically distributed popularity and file sizes. Popularity and sizes of files are not correlated. The same test is conducted for 10 runs with numpy random seed 1 and average results are taken.

```
Hit Penalty:  562.7095685212398
Replacement Penalty:  0.0
Total Penalty:  562.7095685212398
Miss rate:  0.165087
Hit rate:  0.834913
```

Figure 1: Cache Decision Part 1 Results

E. Weak Points

There are some weak points of the algorithm. With this algorithm, the files, which are decided to be cached are not cached until they get requested for the first time. So until all high priority files get cached, there is enough space for caching less priority files. However, these less priority files are not cached even if cache is empty. This algorithm decrease

the overall hit rate but it avoids lots of additional calculations and cache operations resulting in saving running time.

III. Part 2

A. Literature

Cache replacement algorithms are widely used in cache management of network systems, operating systems and database managements. With efficient cache replacement strategy, user experience can be improved significantly by reaching higher hit rates.

Cache replacement strategies are mostly based on recency and frequency properties of data request pattern. Some examples of these strategies are described in following paragraphs.

LRU (Least Recently Used) policy considers recently accessed data is likely to be requested in near future. If cache capacity is full, LRU removes least recent data in cache and stores incoming data. LRU has important role in caching history due to its practical implementation capabilities and O(1) time complexity.

LFU (Least Frequently Used) policy considers frequently accessed data is likely to be requested in the future. When cache capacity is full, it removes least frequently used data in cache and store incoming data. Main drawback of LFU is that LFU cannot adapt new data request pattern if frequencies of data requests are changed.

In order to avoid LFU's incapability of adapting new request patterns, Window-LFU policy is proposed. Window-LFU makes replacement decisions based on frequency as same as LFU but it considers recent past recorded in a sliding window.

LRFU is another policy which considers both recency and frequency with tunable parameter $\lambda \in [0,1]$.

ARC (Adaptive Replacement Cache) policy keeps track of both frequency and recency of accessed data by using 4 Queues. It achieves adaptation to changing request patterns and provide convincing hit rates.[1][2]

B. Caching Approach

In part2 algorithm, since there is no change in popularity of files, there is no need for adaptation to new request patterns and LFU would give satisfying results. Hence, properties of file IDs, file sizes and number of requests for each file are recorded while files were being requested.

Caching algorithm is divided into two parts as before and after convergence. Convergence status is decided belonging to time and number of different files which have been requested.

Before convergence, files are cached and replaced with respect to request number of file.

After convergence, it is accepted "num_ref" queue represents popularity of files similarly to part 1. This queue is used to decide if files are going to be cached or not. Then, all cache is updated according to this decision.

C. Code

Code is written in Python and attached to this report.

D. Results

Caching algorithm is tested for 100000 file requests which have geometrically distributed popularity and file sizes. Popularity and sizes of files are not correlated. The same test is conducted for 10 runs with numpy random seed 1 and average results are taken.

```
Hit Penalty:  623.5711048886994
Replacement Penalty:  0.0
Total Penalty:  623.5711048886994
Miss rate:  0.298337
Hit rate:  0.7016629999999999
```

Figure 2: Cache Decision Part 2 Results

E. Weak Points

The algorithm used before convergence significantly increases replacement amount in cache because popularity of files are not well known at the beginning of run.

Complexity of caching and replacement, increased run times.

It was difficult to decide convergence conditions.

IV. Part 3

A. Caching Approach

In part 3 algorithm, in addition to decrease hit penalty one of the main constraint was to implement simple, practical and efficient algorithm preventing cache replacement as much as possible.

Since there is no change in popularity of files, there is no need for adaptation to new request patterns. Hence, ARC is not chosen. However, T1-T2 queues and Fixed Replacement Cache which are explained in [1] inspired part 3 algorithm.

Three queues are used to buffer incoming requests. With this method, we prevent from caching randomly requested non-popular data. Data files are cached in sequence with respect to their third request timing. Data file, which is requested sooner for three times, is cached earlier as long as there is enough space in cache.

B. Code

Three queues' sizes are adjusted as 100 due to number of files.

Code is written in Python and attached to this report.

C. Results

Caching algorithm is tested for 100000 file requests which have geometrically distributed popularity and file sizes. Popularity and sizes of files are not correlated. The same test is conducted for 10 runs with numpy random seed 1 and average results are taken.

```
Hit Penalty:  554.3640610555786
Replacement Penalty:  0.009999447457041392
Total Penalty:  554.3740605030357
Miss rate:  0.179521
Hit rate:  0.820479
```

Figure 3: Cache Decision Part 3 Results

Results shows that replacement penalty is very low and it is inevitable due to storing files as long as cache capacity allows.

Also hit penalty are lower than part 2 which proves that three queues achieved sequencing data file requests with respect to their popularity.

V. Conclusion

This report contains developments for part 1 and part 2 of the project. Literature, solution approach, optimization of the code, Python code fragment, results and weak points of the algorithms are given in the report.

It can be resulted that it is challenging problem to predict pattern of files to be requested in future. Without knowing popularity of files beforehand. However, by keeping record of requests, one can estimate request pattern and obtain satisfying results.

In this project, popularity of files does not changed within run time. Hence, no real time adaptation is needed for this scenario and algorithms are designed with respect to this projects' constraints.

References

[1] N. Megiddo and D. S. Modha, "ARC: A Self-Tuning, Lowoverhead Replacement Cache" in Proc. 2nd USENIX Conference on File and Storage Technologies, San Francisco, CA, 2003.

[2] Sen Bai, Xin Bai, and Xiangjiu Che. 2016. Window-LRFU: a cache replacement policy subsumes the LRU and window-LFU policies. *Concurr. Comput. : Pract. Exper.* 28, 9 (June 2016), 2670–2684.