Improving Cooperative Caching Using Importance-Aware Bloom Filter  
Lohit Velagapudi  
Advisor: Dr. Hans-Peter Bischof

1. Abstract

This paper describes the working of existing cooperative caching algorithms like Greedy Forwarding, N-Chance and Robinhood algorithms. In this paper, a comparison of these three algorithms is shown and the reasons for an algorithm to work better than other algorithms is presented with results. A drawback of the existing algorithm which could make the cooperative caching process slow has been discussed in this paper and a solution using Importance-Aware Bloom Filter has been proposed. The global cache that lets the clients collaborate could lead to a performance bottleneck and this paper proposes an idea using which the tasks performed by the global cache could be distributed using an Importance-Aware Bloom Filter.

2. Introduction

The need for cooperative caching has arisen because the processor speed is increasing drastically compared to the performance of the disk. This actually means that the processor is able to handle and process data faster than the disk being able to provide data for processing. The performance of the disk is a bottleneck, in this case and to improve this performance bottleneck the idea of caching has come up, which stores the data in the main memory. The storing of data in the main memory improves upon the performance bottleneck of the disk, as it is able to provide data to the processor faster. The main memory is also very expensive, so very small amount of data can be cached in the main memory. To counter this limitation multiple clients could collaborate to form a bigger cache together [6].

3. Bloom Filter in Distributed Environment

3.1. Bloom Filter

The Bloom Filter is a probabilistic data structure that supports set membership queries in a memory efficient manner [1]. The Bloom Filter uses an array of boolean type whose bits are set to false and a predetermined number of hash functions are used. A data item is represented in bloom filter by setting the bits to true using hash functions. While checking for a data item we use the predetermined hash functions to check if all the bits generated by hash functions are set to true. If all are true, then the data item may be previously recorded, if any one bit is false, then the data has not been recorded. There is also a possibility of false positives as the values of bits generated by the hash functions may have been set for representing membership of other data. The standard bloom filter does not permit deletion, none of the bits could be reset to false. The standard bloom filter does not work well when the size of the set...
to be represented by the bloom filter is not known i.e. the bloom filter cannot represent a stream of data [1].

It is important to choose the optimal number of hash functions (k) for a bloom filter, as using more or less hash functions could lead to a lot of false positives. The number of hash functions depends on the size of the bloom filter (m bits) and the set (size of set is n) being represented by the bloom filter [1].

\[ k_{opt} = (m/n) \times \ln 2 \]

3.2. Importance-Aware Bloom Filter

The Importance-Aware Bloom Filter (IBF) is an improved Bloom Filter that could represent set membership of a set of unknown size and also permits deletion. The IBF takes into account the importance of a data item while representing the data item. In case of a standard Bloom Filter, the possible values in the array are binary but in case of IBF, we use an integer array whose bits are set to zero initially but could be set from 1 to a predetermined maximum value (M) based on how important a particular data item is. Every time a data is entered, random p bits are chosen in the array and decremented it by 1. This is done because we do not know the size of the set, so if the decrementing is not done then all the bits will have value more than 0 and the Bloom Filter will return more false positives. By decrementing p positions in the array, we are representing only the important data and eliminating the less important data from the IBF. This way the set membership queries would return fewer false positives and false negatives for frequently occurring data items.

There are two type of Importance-Aware Bloom Filters: 2-Class and Multi-Class. The 2-Class IBF has values set to M or M/2 but the Multi-Class IBF has values set anywhere between 1 to M based on the importance of the data. The importance of the data is based on the number of occurrence of a particular data [4].

3.3. Bloom Filter vs Importance-Aware Bloom Filter

The performance of Bloom Filter and Importance-Aware Bloom Filter is compared in a cooperative caching environment, for which every client cache works along with a Bloom Filter. The cache hits in this case are recorded only when the Bloom Filter represents that particular data and the data is present in the client cache. The false positives are recorded when the Bloom Filter represents the data but the client cache does not have that data.

The graph-1 shows the comparison of Bloom Filter and IBF false positives while increasing the client cache size. The goal of this is to understand how well a Bloom Filter or IBF could be used for a cooperative caching mechanism. As the graph-1 shows the number of false positives while using the Bloom Filter is nearly double that of IBF, but the false positive decrease as the client cache size increases. The reason
for having a high false positive for a Bloom Filter initially because it has lesser data due to the size of the cache, but as the cache size increases more data is retained in the cache which leads to less false positives. The false positives for the IBF is very constant throughout because the false positives in case of IBF does not arise due to all the bits being set like in Bloom Filter and also the IBF tries to only represent the important data, so there is less likelihood of false positives in case of IBF.

Graph 1

Client Cache Size vs False Positives

Graph 2

Client Cache Size vs Cache Hits
The graph-2 shows the number of cache hits while using a Bloom Filter and an Importance-Aware Bloom Filter. The difference in the number of cache hits increases as the number of client cache size increases. The reason for high hit rate in case of Bloom Filter is because the Bloom Filter is not able to handle a stream of data, which leads to a lot of false positives. Due to unknown number of data, all the bits are set to true which leads to later requests being false positives and also the data is present in the client cache, so it becomes a cache hit. In case of IBF, only the important data is represented which is the reason for fewer cache hits.

4. Components of Caching Algorithm

4.1. Client Cache

The client cache or the cache is the basic and most important part of a caching process. The client cache is used in the caching process even if it is a non-cooperative caching algorithm. The client cache stores some part of the requested data to serve the future requests faster. The client cache has a limitation of size, so it can only store a certain number of requested data. Whenever a client is requested for a data, it first looks up in the client cache for the data before forwarding the request to other clients or the server.

The client cache could use different algorithms to retain most recently or frequently used data. For the purpose of this project, we use the Least Recently Used (LRU) caching algorithm, which retains the most recently used data and discards the least recently used data. This algorithm works for the project as it makes the recently added or accessed data the most recently used data and does not discard it soon, whereas as a Least Frequently Used (LFU) caching algorithm may discard recently added data sooner, if there are no frequent requests for it after its initial addition.

The LRU client cache is implemented using a LinkedHashMap data structure of Java for this project. The LinkedHashMap is able to provide O(1) access to a requested data using the hash function and is also able to maintain the order of the most recently used data using a Linkedlist.

4.2. Global Cache

The global cache is the key component of a cooperative caching algorithm. It is a central resource that permits the collaboration of the clients to form a bigger cache together. The global cache does not store all the blocks of data present among the client, it just stores the information about which client holds a particular block of data. This way the global cache is able to route the request from a requesting client to the client that holds the data and also the global cache could be space efficient. The size of the global cache is less than or equal to (number of clients * client cache size).

The global cache is handled by the manager. The request to global cache happens only when there is a client cache miss and the global cache misses are handled by the server. The global cache could be implemented by a data structure that stores data in
key-value format, where the block id of the data is the key and the set of clients holding that particular data is the value. The data structure would use a hash function to find a block id in O(1) time and pick the client from the set of clients that holds the data with the block id. The global cache does not store a block id which is not cached with any client.

4.3. Disk

The disk is the main source of all the data which is held by the server. The goal of any request is to get the data from the disk. The goal of the caching process is to serve the requests faster and in a more effective manner without contacting the server. The request to the server for the data from the disk is the most unfavorable stage of a caching process as it is expensive. Since, the disk is the source of all the data, so requests being sent to the server is inevitable but the goal is to minimize the requests to server through caching.

5. Cooperative Caching Algorithms

Caching is a process where the client stores some of the requested data to serve future requests faster, as sending requests to the server is an expensive process. The caching process could use an algorithm that stores the recently or frequently used data. The client has a limitation where it could only store a certain number of requested data. The higher the size of the client cache, the fewer requests would be sent to the server. So, the cooperative caching is a process where multiple clients cooperate with each other to form a bigger cache, this way the clients share the cached data among themselves. The cooperative caching algorithm provides access to a bigger cache to each client, even though the size of the client cache is very small. One of the ways to achieve cooperative caching is to use a global cache. This global cache serves the requests that could not be served by a client cache, by forwarding the request to a client that holds the requested data. The cooperative caching reduces the number of requests to the server and makes the server available to handle requests for data that is not cached with the clients.

This paper discusses three cooperative caching algorithms: Greedy Forwarding, N-Chance and Robinhood algorithms.

5.1. Greedy Forwarding

The Greedy Forwarding algorithm is the simplest cooperative caching algorithm, where each client work like in a non-cooperative caching algorithm, but uses a global cache for cooperation. The clients in this algorithm are not aware of the contents of other client cache and do not maintains its cache based on the contents of other client cache, so each client works in a greedy manner.

In Greedy Forwarding algorithm, when a client receives a request it checks for the data in its local cache and if the data is not present then the request is served by other clients with the help of global cache. If none of the clients have cached the data then the request is forwarded to the server.[6]
One difference in the working of the client in Greedy Forwarding algorithm to a non-cooperative caching algorithm is that it needs to update the global cache and forward requested data to the requesting client. The client needs to add to the global cache when a new data is being added to the client cache and delete from the global cache when the least recently used data is being discarded from the client. The client should also be able to forward data to the requesting client directly to reduce the workload of the manager. One advantage of Greedy Forwarding is its simplicity and not adding to the workload of the manager [6].

5.2. N-Chance

The working of N-Chance is similar to Greedy Forwarding, but the clients in case of N-Chance do not function greedily. The N-Chance clients cooperate with each other to avoid the discarding of singlets. A singlet is a block of data that is only cached with one client. The singlet information is maintained by the manager, which is also responsible for the global cache maintenance. This information is requested by the client to check if a least recently used data being discarded is a singlet. If a data being discarded is a singlet, then this singlet is sent to a random client, and this random client adds the data to its cache. This forwarding of the data to another client is called recirculation and each block of data has a predetermined recirculation count. The recirculation count is the number of times data could be forwarded to another client. The recirculation count of the data is decremented whenever the data is forwarded and once the recirculation count is 0 then the data is discarded as if it was not a singlet. The reason for calling this algorithm N-Chance is that a singlet is given N number of chances for it to become a non-singlet before discarding it [6].

The difference between N-Chance and Greedy Forwarding is the manager maintaining information about singlets by keeping a count on the number of occurrences of a particular data among the clients and updating it whenever there is an update to the global cache. The clients in N-Chance also have to choose a random client to forward a singlet to other clients, so each client has to have access to the information about the other clients in the system.

5.3. Robinhood

The Robinhood algorithm differs from N-Chance in the way it tries to preserve singlets. The Robinhood algorithm rather than sending a singlet to a random client, it identifies a client using a specific algorithm to send the data. The data here is not simply added to a client to which the data is forwarded, but it also identifies a data in the client to be replaced. The client to which the data is forwarded is called a victim client and the data that is replaced at the victim client is called victim block. The victim block is the block of data that is cached at multiple clients and these multiple clients are called victim clients. The singlet in case of Robinhood algorithm is forwarded to one of the victim clients. The information about the victim client and the victim blocks is managed by the same data structure that helps identify a singlet.
In case of N-Chance the singlet replaces a least recently used block of data at the random client whereas in case of Robinhood the singlet replaces a victim block that is cached at multiple clients. Since, Robinhood algorithm is looking for clients caching rich blocks of data, so the victim clients are always active clients but N-Chance could forward singlets to an inactive client. The advantage of Robinhood over N-Chance is that it may not lead to a chain of singlets being forwarded to other clients, since it is replacing the rich block with a singlet and not discarding a block of data [7].

6. Evaluation:

6.1. Simulation Framework

The cooperative caching algorithms described in this paper need to be evaluated and compared to understand their working better. For this purpose, a simulation framework is designed to simulate the working of the caching algorithms and keep count of access to a cache or a disk. The software simulator would have a class of Server, Manager and Client that would replicate the behavior of real world servers, managers and clients. The simulator would have a Simulator class that takes inputs from the user and will set up the experiment by adding data to the cache. At the end of the experiment, Simulator writes to a CSV file the number of cache hits, disk accesses, false positives, etc. for evaluation.

![Software Simulation Framework](image)

The different components of the software simulator are as shown in figure 1 could be described as follows:

- Server: The server handles the disk. The disk stores data in the form of blocks. The size of the disk is predetermined.
- Manager: The manager is responsible for managing the global cache. The global cache stores all the data stored by all client caches and is aware of which client
holds a particular block of data. The global cache permits the cooperation of clients.

- Client: The client receives a request for the data which in case of simulation is read from a file. The requests are the block identifiers. The requests are served by client cache. If the data is not cached locally, the request is forwarded to global cache.

6.2. Parameters:

6.2.1. Cache Hits & Disk Access

One of the goals of cooperative caching is to have maximum cache hits and minimum disk accesses possible because disk access is expensive. This desired effect in case of cooperative caching can be achieved if the cache with the clients could hold all this data and this could happen if each client holds more data or there are more number of clients. Ideally, there would not be any disk access if the cache size of all the clients together was greater than or equal to the size of disk. This parameter gives an indication if the goal of cooperative caching is achieved or not. [6][7]

6.2.2. Block Access Time (Ticks)

The tick is a measure of movement of a block of data in cooperative caching system. The disk access ticks are the most expensive ticks, as disk access is an expensive and slow process. This is also one reason that makes the disk access unfavorable. The best and the minimal tick is the local cache tick and the next best is the global cache tick, through which the data is retrieved from other clients. The global cache tick is expensive compared to the local cache tick because there is network usage to get a block of data. [6][7]

6.2.3. Unique Blocks Cached

This parameter is an indication of the number of data blocks cached with the all the clients. The goal of cooperative caching is to have all the data present in the disk among the clients, this way there would not be any disk access. This could only happen when the size of the all client caches together is at least equal to the disk size. This parameter acts like an evidence to the disk access or cache hits happening in a certain experiment. [6][7]

7. Comparative Analysis of Caching Algorithm

The existing caching algorithms are compared in two ways:

1. Increasing the client cache size while keeping the number of clients constant.
2. Increasing the number of clients while keeping the client cache size constant.
The comparison is based on the parameters like cache hits, disk access, block access time and number of unique data among the clients.

7.1. Cache Hits & Disk Access

The graphs 3 and 4 show cache hits and disk access that are recorded while increasing the client cache size from 10 to 510 while keeping the number of clients are 10 and disk size is 1000. The cooperative caching algorithm should have 0 disk accesses when the client cache size is 100. But the above graphs shows some disk access when the client cache size is 100 that is because this experiment is not using perfect traces. The perfect traces are non-redundant traces such that all the data would be present in the client caches and there would not be any disk access. The real world does not have a perfect traces, hence the above results resemble a real world cooperative caching process. [6][7]
The results shown in graphs-5 and 6 are recorded while increasing the number of clients from 10 to 310 and keeping the client cache size constant at 10 and the disk size of 1000. In the above graphs, Robinhood algorithm has the best cache hit and disk access rate followed by N-Chance and Greedy Forwarding respectively. The reason for Robinhood and N-Chance working better than Greedy Forwarding is because the clients in these algorithms cooperate with each other to prevent discarding of singlets. The Robinhood algorithm has a better singlet retention algorithm compared to N-Chance. The N-Chance algorithm randomly adds a singlet being discarded to another client cache, which could lead to another singlet to be discarded at other client and this may continue till a client has a non-singlet being discarded. So, the N-Chance singlet retention policy could lead to a chain of singlets being transferred to other clients, this leads to data blocks losing their recirculation count sooner and getting discarded. So, the Robinhood algorithm has more data among clients and hence more cache hits compared to N-Chance. [6][7]
7.2. Block Access Time

The graphs 7 and 8 show the relation between increase of the client cache size and the number of clients with the block access time. In both the cases, the block access time reduces because the block access time is maximum when the blocks are accessed from the disk, but as the size of the client caches together increases, there are fewer requests to the disk.

If you compare the above two graphs, the block access time is lesser as we increase the client cache size as compared to while increasing the number of clients. The reason for this is as we increase the client cache size, the number of requests served by other client caches also decreases, which is not the case while increasing the number of clients as the client cache size remains constant. [6][7]
7.3. Unique Blocks Cached

The graphs 7 and 8 are a representation of the number of unique blocks cached among the clients as the number of clients or the client cache size increases. The goal of the cooperative caching algorithm is to have all the data in the disk available among the clients, when the clients together have enough space to store the data. If all the data in the disk is present among the clients, then all the requests are cache hits and there are no disk access. In the earlier graphs, the cache hits are not 100% when the number of clients or the client cache size is 100 because not all the contents of the disk are stored among the clients when the number of clients or disk size is 100. [6][7]
8. Proposed Algorithm

8.1. Drawbacks of Existing Algorithm

The global cache is a central resource in a cooperative caching algorithm, that enables the cooperation among the clients and it is an accurate representation of the contents of the all the client caches together. So, the global cache has to be frequently update when there is an addition or deletion to the client cache to maintain the global cache as an accurate representation of all the client caches. As the number of clients increase, the number of update requests to the global cache also increases. The global cache also has to perform the operation of checking if a particular block has been cached among clients. The global cache is pivotal in the cooperative caching process and due to the high number of requests to the global cache, it may reduce the availability of the global cache which may slow down the caching process and the clients may have to wait to make an update to the global cache or to make a request.

8.2. Solution

The drawback of the existing algorithms is the increasing number of requests to the global cache as the number of clients increase. The reason for such high number of requests is because the global cache is an accurate representation of the contents of the client cache. One solution to resolve this issue is to have a less accurate version of the global cache that has fewer update requests compared to the original global cache. This less accurate global cache uses an array of Importance-Aware Bloom Filters (IBF) with each IBF representing the cache of the client. These array of IBF would be handled by the IBFManager.

The figure-2 describes the framework of the proposed algorithm. The IBFManager would receive the client cache misses and the IBFManager would check the array of IBFs for a client caching the particular data and forwards the request to the client caching the data. The IBF is a probabilistic representation of the client cache and represents only the most frequently requested data items (i.e. important), the less frequently requested data items are not represented by the IBF and those requests are forwarded to the Manager. The global cache represented by the IBF is updated at the IBFManager at periodic intervals and the accuracy of the IBF is not affected to a great extent as it is a probabilistic representation of the client cache [5].
As the figure-2 shows, the number of requests received by the Manager would only be added or deleted to the global cache. This way the block requests are sent to the IBFManager and if the IBFManager fails to fetch a data the request is then forwarded to the Manager. The global cache maintained by the IBFManager using the IBFs of the client cache, is a less accurate version of the global cache maintained by the manager. The reason for less accuracy of the global cache with the IBFManager is because IBF is a probabilistic data structure and also the IBFs at the IBFManager are only updated at frequent intervals and not regularly.

9. Results

The graphs 11 and 12 shows the results of the proposed algorithm whose goal is to reduce the manager load. The graphs clearly state that the number of requests that are being handled by the manager reduces as the client cache size increases or the number of clients increases. The requests that are received by the manager are forwarded by the IBFManager only when the IBFManager is not able to handle the requests from the client. Most of the requests that are forwarded to the Manager are for the less important data which the IBF does not represent. The requests in the graphs do not show the update requests, but only the block requests.
10. Future Work

The goal of the proposed algorithm is to reduce the work load of the Manager which enables the cooperative caching process. The IBFManager chosen for this purpose uses multiple Importance-Aware Bloom Filters (IBF) that serves the purpose of this project. There is a scope to further improve the design of IBFManager such that the finding of the block is as fast as looking up in a global cache. For this purpose, the idea of hint-based cooperative caching could be used which maintains less accurate hints that reduces the number of requests to the global cache [5].
11. References


DOI=10.1145/362670.362675


[7]. Eric Anderson, Christopher Hoover, Xiaozhou Li “New Algorithms for File System Cooperative Caching”