Improving Cooperative Caching Using Importance Aware Bloom Filter
Pavan Kumar Pallerla
pp4475@rit.edu

ABSTRACT
This paper discusses cooperative caching in distributed systems. Caching is used in any application to reduce latency, shorten access time and improve I/O. Caching in distributed context is also used to reduce dense communication between clients and server along with the reasons mentioned above. Cooperative caching does this by sharing cache among clients. There have been many algorithms proposed on cooperative caching and the paper discusses three of them from the literature. These algorithms are evaluated in terms of disk accesses rate and the accuracy of cache hit rate. A proposed algorithm is also discussed which is assessed using the same parameters. Scalability of all the algorithms is also verified using a different number of clients and cache sizes. A simulation framework is developed to implement the algorithms and test their performance.

1. INTRODUCTION
Cooperative caching is used in any distributed file system to reduce latency, shorten access time and improve I/O. It is widely used in data sharing applications over the network to reduce bandwidth usage and latency. But caching by itself does not have the knowledge of distributed network. This is where cooperative caching is used to bring those two concepts together. Cooperative caching uses some memory from each client to create a cumulative cache for a centralized file server. We call the cumulative cache as global cache in this paper. The global cache is used to deliver requested content in part by sharing caches between clients. When the global cache fails to provide for a data request from one of the clients, only then the server is accessed. Reducing the number of accesses to server is the objective of cooperative caching. Building a global cache requires a central manager which can redirect client requests. The central manager has to keep track of cache changes of all the clients. This process offers a lot of overhead and is memory intensive. Usage of Bloom filters to keep the manager up-to-date of the client caches is an efficient solution to this problem. There is an improvised version of Bloom filter called Importance Aware Bloom filter which takes into consideration the importance of the element being inserted into it. Our paper compares the performance of cooperative caching algorithms that are built using Bloom filter and Importance Aware Bloom filter.

In the next sections we discuss the Bloom filters in detail. We will explain about the simulation framework that has been designed to evaluate the algorithms, the algorithms that are implemented, implementation procedure, data structures and techniques used. We have also done graphical analysis of simulation studies.

2. LOOK-UP TABLES
2.1 Bloom filter
Bloom filter is a probabilistic data structure conceived by Burton Howard Bloom [1]. It is used to test whether an element is part of a set. It offers a probabilistic way to represent a set that can result in false positives. This means sometimes Bloom filter tells that an element has been inserted when it is actually not. But, Bloom filter does not give out false negatives. That means when an element is inserted Bloom filter does not say that the element has not been inserted. The basic operations involved in a Bloom filter are adding elements and querying for them. A basic Bloom filter only allows addition of elements but
not removal. Because of this nature of the Bloom filter, the more elements added to it increases the probability of false positives to queries.

Implementation of Bloom filter is done using a bit array of m bits. All the m bits are initially set to 0. A set of k different hash functions are used in it. For a given element x, the hash values of k hash functions are computed. These hash values are less than the size of bit array m. When an element is to be added, the array positions at the hash values are set to 1. Whereas to query if an element is present, the bit values at array positions of hash values are checked if they have already been set to 1. If there is at least one position that is not set to 1, then it is assumed that the element x is not inserted in the Bloom filter. In a case where all the bits in the hash value positions are set to 1, then the element x is probably inserted. It is possible that while inserting elements other than x, the bit positions corresponding to x could have been set to 1 and this results in false positives.

The key to avoid false positives in a Bloom filter is selecting appropriate number of hash functions, size of the Bloom filter and number of elements added to the set. To insert n different elements in a Bloom filter of size m, an appropriate value of k in order to reduce false positives is given as \( k = \left(\frac{m}{n}\right) \ln 2 \).

We have unknown values of m and k here so, we have fixed the number of hash functions to 3 and n is measured in each experiment to calculate the appropriate size of the Bloom filter. Choice of hash functions is very important so after careful consideration Jenkins and Murmur hash functions have been chosen. The reason behind the choice is that these functions return wide range of integer values which can be easily converted to a value \( 0 \leq h(x) \leq m \). Where as the other popular hash functions like MD5 return a hash value of 16-byte which requires additional computation of converting into integer value. Keeping performance of the framework in mind, hash functions which return integer values have been chosen. The third one is a pseudo hash function \( f(x) = h(x) + g(x) \). A number of Bloom filter variants have been proposed to address some of the limitations of of the original structure, including counting, deletion, multisets and space efficiency. In the next section, we examine the Importance Aware Bloom filter variant.

### 2.2 Importance Aware Bloom filter

Many applications assume the popularity of data items and queries to be identical. In contrast they are varied and skewed. In many networks data popularity is observed to be similar to Zipf’s [2] distribution. Zipf’s distribution is an empirical law which states that the frequency of any element in a corpus is inversely proportional to its rank in the frequency table. The standard Bloom filter explained in the previous section does not take this fact into consideration while adding datablocks. An intuitive approach to take data popularity into account has been proposed in Importance Aware Bloom filter (IBF) [3]. Unlike standard Bloom filter, IBF also supports eviction of elements. Since it supports eviction too, false negatives are also possible in IBF. The basic idea behind IBF is to store important data elements for longer period of time. In other words, datablocks with lower importance are selected for eviction. Thus IBF results in lower false positives and false negatives for data with high importance. At the same time data items with lower importance also has reasonably lower false positive and false negative rates. This phenomenon has been proven experimentally and attributed to less number of requests for lower importance data items.

A standard Bloom filter is an array of bits but an IBF is an array of integers. Usually a preselected d bits are allocated to represent each element which is called a cell in the array. Value of d is selected as 3 for implementation based on the empirical analysis made in [3]. An Importance function is used to determine the importance of a data item. For any data item it returns a value of 0 to a value \( M \), where
\[ M = 2^d - 1. \] According to the previous equation, \( M = 7 \). Each datablock cached at client has a \textit{accessCount} variable and is used to count number of requests occurred on the datablock. It has a predetermined value \( M \) and \textit{accessCount} is not incremented beyond \( M \). The \textit{accessCount} serves as the importance function here. When a datablock is being added to IBF its \textit{accessCount} is checked for importance. If it obtained from the server, the \textit{accessCount} is set to 0 by default whereas if it is forwarded from another client the \textit{accessCount} is incremented by 1 by the forwarded client and sent to the requesting client. Similar to Bloom filter, IBF also uses \( k \) hash functions to add an element. Before addition, IBF checks for duplicate insertion by making sure all the \( k \) cells in the array are not zero. Then a random \( P = 7 \), also selected from [3], cells are decremented by 1. This is done to make way for future additions to IBF, else it will be flooded. Then the \( k \) cells are set to importance value of the datablock.

The authors in [3] prove that after a large number of iterations, there will always be a fixed fraction of cells in IBF with zero value to accommodate new insertions irrespective of \( M, k \) or \( P \) values. There are a few variations of IBF: IBF-2C read as Importance Aware Bloom Filter Two Class and IBF-Multi Class. In IBF-2C, when inserting element in IBF, a value of \( \frac{M}{2} \) or \( M \) are added to the cells based on the importance of data item. Whereas in IBF-Multi Class value of addition is \([0, M]\). In our framework we are using IBF-Multi Class.

3. RELATED WORK
There has been a lot of work done in the field of distributed cooperative caching such as Summary Cache [4], Hint Based Caching [5] and Cooperative caching in P2P networks [6]. The basic idea of our simulation framework has been inspired from Summary Cache algorithm. In Summary Cache, there are multiple proxies present. Each proxy has what they call a “cache directory” for all the other proxies, including itself. Cache directory is a summary of caches at clients. Thus the name Summary Cache. These cache directories are not maintained up-to-date. They are either updated at regular intervals or when a certain percentage of cached data are not reflected in the cache directory. A client requests its own local proxy for any data item. The proxy first searches its own cache directory. If the summary data returns that data item is present in one of the clients, it forwards the request to that client. If the data item is indeed present in the client cache, it is forwarded to the requested client. Else the client responds to the proxy saying that it does not cache the data item. In this manner the proxy looks for a data item in its own cache directory first. If it does not find the data item, it goes on to search for it in the summary of other proxies. Forwards the request to another proxy if it finds the data item in it. The other proxy which caches the data item forwards it to the requesting proxy and not the client directly. If it is not present in any proxy, then the request is sent to the server. Summary Cache has been proposed for web proxies to reduce the web traffic and alleviate network bottlenecks. It has been designed as an alternative for other cooperative caching algorithms designed for web proxies like Internet Cache Protocol (ICP) which multicasts query messages to neighboring proxies.
Most of the cooperative caching algorithms have a central manager which is responsible for the maintenance of global cache that keeps track of cache data of the clients. In a system that is heavily dependent on the manager, clients keep the manager updated by sending an update each time a datablock moves in and out of their caches. This increases overhead on the manager. Hint based approach [5] proposes a solution where this overhead is aimed to reduce by distributing the centralized control of global cache. It is done by distributing portions of global cache among the clients as hints. This helps clients make caching decisions like forwarding block requests and relocating the blocks. Although manager retains control over the global cache, it receives much less overhead from the clients.

In short, global cache is maintained by both the manager and clients together. Hint based algorithm uses global cache to maintain the information about the location of a master block. The master block locations are stored as facts and hints. The facts are stored in the manager and hints are stored in the clients. When the block is fetched for the first time from the server and cached by a client, it is designated as a master block. There can only be one master block at any given time but, there can be many copies of master block in the clients. It is illustrated in the figure 3. The main idea behind introducing hints is to reduce dependence on the manager for master location. But this is done at the cost of getting inaccurate hints. Inaccuracy occurs because the hints are local to the client and do not reflect the changes occurred on the caches of other clients. For example, when a hint tells that a master block is present at a client there is a possibility that the master block is not present. This scenario arises when the master block has been forwarded or evicted from the cache altogether and hints are not updated to reflect this change. In this algorithm clients use hints to obtain a data item. A client first requests the client that is referenced in the hint for the data item. If the client actually has the master block, it forwards the data item. But if it turns out that the hint is inaccurate as it has been explained in the previous paragraph, the client contacts the manager for the fact and updates its own hint for that particular block. For the manager to always have accurate facts, all the clients when they evict or forward a master block they update the manager with the new location of the block. Although hints introduce less overhead on the manager, as the clients increases the communication between manager and clients increase. This is because more the clients more the obsolete hints and it results in more communication between manager and clients. This hinders scalability of the algorithm.

![Fig. 3. Masters copy in Hint Based [5]](image-url)
Another type of cooperative caching algorithm was proposed for peer-to-peer (P2P) networks. This algorithm [6] is specially designed for the push based networks where server has to broadcast many kinds of data in order to satisfy all the clients’ demand. Since it is a push based system, the clients have nothing but to wait for their required data to be broadcasted. This causes a lot of waiting in the clients’ part. The algorithm aims to reduce the waiting time by caching the data pushed by server at the clients. The clients listen to the broadcasted data and also access their own data. A client first tries to access the data from its own cache. If it fails to obtain the data item from its own cache, it floods request to its own peers. The peers then search for the data in their caches and if found, they respond back by sending the data item to the requesting client. If a peer does not have the data item in its cache, it in turn floods the request to its peers. In this way the flooding continues until either the data is found or a predetermined time to live (TTL) is reached. The algorithm also proposes a novel approach to reduce the flooding in the network. It is by the providing the peers the ability to guess which peer might have the desired data item. This is achieved by keeping track of the fate of the queries passed through a peer. Each query can end up either in \textit{Connected, Successful or Failure}. Connected being the query has been served by the peer. Success status for a query is obtained when a peer or further peers has served the query request. A query is marked failure when none of the peers or their peers could serve the query. Figure 4 illustrates the query propagation among peers.

![Query propagation](image)

\textbf{Fig. 4. Query propagation [6]}

\section{ALGORITHMS}

This section examines three variations of cooperative caching algorithms in detail. The reason for choosing these particular algorithms is that one algorithm builds up on the demerits of the other algorithm and thereby achieving the best possible results in cooperative caching. The following sections cover these algorithms in ascending order in terms of hit ratio.

\subsection{Greedy Forwarding}

Greedy Forwarding [7] is a simple cooperative caching algorithm. In this algorithm each client’s cache resource is treated as an integral part of the global cache. The reason behind such consideration is to satisfy individual client’s datablock request by accessing the global cache. As the name of the algorithm suggests, each client greedily manages its local cache without any regard to the other clients’ datablock requests and their contents. In this algorithm when a block is not found in the local cache of a client, it sends a block request to the server. In our implementation, the request is sent to the manager which centrally manages the global cache. If the block is present in the memory cache of the server, it forwards...
the block to the requested client. Otherwise, the server looks up for the block in a data structure which keeps track of datablocks present in all the clients. In our implementation this data structure is a Bloom filter. Manager maintains a summary of each client using Bloom filter assigned to it. Greedy Forwarding does not mandate the usage of Bloom filters. But considering the efficient memory usage and performance, Bloom filters are chosen for the task. Management of Bloom filters are discussed in detail in the framework and implementation sections. If the server finds any client that caches the datablock, it forwards the request to that client. Then that client sends the datablock to the requested client directly. The response is not sent via the server in order to avoid the unnecessary latency and server’s workload. In the case where no client has the requested datablock in their caches, the server fetches it from the disk and forwards it to the client.

The only change to the algorithms that were proposed prior to Greedy Forwarding is that the server needs to be able to forward requests to other clients and the clients need to be able to handle the forwarded requests. Also, the clients keep the server updated of the contents in their caches. Because of the Bloom filter’s nature to give out false positives for membership queries, a forwarding client might not actually have the requested datablock. In such a case, the client responds to the server indicating the same. Then the server goes on to search for the datablock in other clients. The additional memory and request processing overhead justified because it reduces the disk accesses by a great deal. This is a desirable attribute because disk access is considered a much more valuable resource as compared to the overhead. The attribute of clients managing their local cache greedily while forwarding the requested datablocks to other clients is appealing. But lack of coordination among the clients lead to data duplication among clients. Thus not effectively utilizing the available limited memory resources. Greedy Forwarding forms a basis to a set of cooperative cache algorithms. N-Chance Forwarding and RobinHood are a few among them that address the issue of client coordination to effectively utilize the resources.

4.2 N-Chance Forwarding

As mentioned in the previous section N-Chance [7] attempts to improve on the demerits of Greedy Forwarding [7]. It introduces coordination between clients by trying to cache singlets for a longer time in the global cache. Singlet is defined as a datablock that is cached only at one client cache. The idea behind keeping the singlets in the global cache for long time is that it increases the total number of distinct datablocks present in the global cache at any given time. It is achieved by avoiding discarding singlets from client cache. When a client discards a datablock from its cache, it checks whether the datablock is the last copy cached by any client. In our implementation this is done by contacting the manager. Manager keeps track of how many clients cached a particular datablock. If the datablock about to be discarded is indeed a singlet, then instead of discarding it, the client sets the datablock’s recirculation count to n and forwards it to a randomly selected client. That client receives the singlet and adds it to its cache. Since the client’s cache is constantly modified and the singlet reaches the end of the cache and about to be discarded once again, now its recirculation count is decremented by 1 and forwarded to another client. This process continues until the recirculation count of the singlet reaches 0. Thus gives the name N-Chance. Meanwhile if a client requests for the singlet and the client forwards the requested singlet. Since now that there are multiple copies of the datablock, it stops being a singlet and its recirculation count is reset.

Greedy Forwarding is nothing but an N-Chance Forwarding algorithm with recirculation count n = 0. The algorithm provides a dynamic trade off between client’s local data, data being cached because client referenced it and global data, singlets being cached for the good of aggregate system. There is a possibility of ripple effect occurring because a singlet is forwarded to one client and it in turn forwards a data item to another client and so on. This has been prevented in the implementation by not allowing a client to forward a singlet when it receives one. In such case a modified replacement algorithm takes
effect which discards the least recently used block. The main advantage of N-Chance is that it provides a simple dynamic trade-off between each client’s private cache data and date being cached for the good of the overall system. Storing singlets for longer time in the system provides a better performance than Greedy Forwarding. Although it can be perceived that there is an increased system overload from singlets bouncing from one client to the next, the hit rate of data items shows significant improvement.

4.3 RobinHood
RobinHood [8] algorithm is based on the N-Chance [7] algorithm. In N-Chance when a datablock is being evicted, client asks the manager if the datablock is a singlet. If it not a singlet, the datablock is simply discarded. Otherwise, the singlet is forwarded to a random client to cache at the top to client cache. To prevent the singlets from circulating in the system forever, a recirculation count is introduced and each singlet is forwarded only those many number of times.

There is an uncertainty in which client the singlet is forwarded to since the forwarded client is randomly selected. RobinHood introduces a concept called victim client. Instead of randomly selecting a client to forward the singlet to, such a client is selected which has the victim chunk. A victim chunk is the datablock which is cached at many clients. Forwarding client obtains this information from the manager.

The manager does this by maintaining number of clients in which each datablock is cached. If the manager cannot find a victim chunk, then RobinHood performs as N-Chance. The huge advantage of RobinHood over N-Chance is that the former chooses such a victim chunk that its removal does not have any adverse effects on the system but improves it by making space for a singlet.

4.4 K-Sequence
The underlying idea of K-Sequence is that a file system uses contiguous blocks of data. Taking that into consideration, it makes sense for the clients to maintain datablocks in sequence. The number of datablocks that form such a sequence is represented by K, thus the name KSequence. The value of K is provided at the beginning of the experiment. The experiments have been run for a K from 1 to 10.

Results of those experiments for all the K values are compared. This algorithm works similar to NChance, but attempts to keep sequences in the system. The only difference between NChance and KSequence is that the former considers a datablock which is cached at only one client as a singlet whereas in the latter, a whole sequence of blocks which is only cached at one client is considered as a singlet.

Eviction occurs in KSequence in a very similar manner as that of NChance. When a block is being evicted, it is checked if the block is part of a complete sequence. If it is and there is only one such sequences available among all the clients, then the complete sequence is evicted and forwarded to a random client. That is the only similarity between NChance and KSequence. There is another unique functionality that only KSequence has and that is: when a client requests a datablock, it is more likely to request the other datablocks of the sequence to which the requested datablock belongs to. So, if another client has the requested datablock’s sequence in its cache, then it sends the whole sequence to the requesting client.

5. IMPLEMENTATION
A simulation framework has been designed to implement and compare performances of all the four algorithms. It is designed in such a way that all the algorithms are accommodated at the same time. The framework has been developed using Java 1.7 on Mac OS X El Capitan. The implementation is generic to all kinds of operating systems.

The architecture consists of client objects, a proxy object which is called manager and a server object. The manager and server objects are singletons because the real world applications will have only one manager to service requests of all clients and only one server to serve the needs of each client.

Execution of the simulation application is called an experiment. Every experiment is used to analyze the performance when one particular evaluation parameter’s value is varying (usually increasing with a
predetermined difference). Various evaluation parameters and their purposes will be discussed in the Evaluation Parameters section. Inputs for the framework is provided using a configuration file which is named a config file. The purpose of a config file is that all the inputs can be read from it instead of providing command line parameters for each input parameter. It helps to read and provide the inputs in a systematic manner. The parameters that are provided using config file are the kind of Bloom filter used in an experiment, Algorithm used in the experiment, disk size of the server, minimum value of the experiment variable, maximum value of the experiment variable, difference with which the experiment variable has to be increased, cache size which is equal for all the clients, number of clients used in the experiment. Except for the experiment variable, all the other parameters will be constant through out the experiment.

The reason for using a minimum, maximum and difference of experiment variable is to execute the experiment with a range of values for experiment variable without having to manually execute the application every time with a different value. In an experiment the hit count and disk access count for a particular experiment variable value is printed out on to the console for further analysis. When a client has forwarded a datablock from its cache to another client to serve its request is called a hit. Multiple clients will serve other clients’ requests in an experiment and the number of hit occurrences are counted and it is called hit count. When none of the other clients can serve a request from one client then the request will be served by the server itself and it is called a disk access. The number of occurrences of a disk access is also measured and is called disk access count. Later the hit counts and disk access counts are plotted against the experiment variable values to analyze the behavior of the algorithm.

The entry point for the experiment is the Simulator. It reads the config file which is a properties object and reads the input parameters. The above mentioned input parameters are absolutely necessary for the experiment to execute. If anyone of them is missing in the config file or any parameter has formatting or type error, the application exits. In such cases the user will have to intervene and fix the errors to continue. Once all the input parameters are read successfully, the Simulator initializes the server and the manager objects. There are two kinds of manager objects, one which uses standard Bloom filter and the other that uses Importance Aware Bloom filter. The manager that gets initialized depends on the value provided for BloomfilterType input parameter in the config file. The manager has an array of Bloom filter or Importance Aware Bloom filter based on the input parameter provided. The size of the array is the number of clients in the experiment. This array of Bloom filters is to represent the cache present in each client. Once the manager object is initialized the Bloom filter array is created in the manager. After the manager and server are initialized, the client objects get initialized. An array of clients is maintained.

Once all the clients, manager and server are initialized, the data is distributed to the client caches. The data is distributed randomly to each client, using Random object in java.util. Any given client will not have a datablock twice in its cache. Because the Random object is used, a given datablock can be present in one or more clients. The randomly generated data cannot be greater than the disk size of the server. A separate array list is maintained to represent all the data that is allocated to the clients altogether. As a data is allotted to a cache in a client, it is also saved in the array list. This array list is later provided to each client as a list of requests to be made by each client. A client goes through this request list one datablock at a time and first checks if the datablock is present in its local cache. If it is present in the local cache, then it goes on to check the next datablock. On the other hand, if the datablock is not present in the local cache, it sends a request to the manager. The manager iterates through the array of Bloom filters it has to until it finds the client which cached the particular datablock. If none of the clients have it in their local caches, the manager retrieves the datablock from the server. If the manager finds a client which might have the datablock, then it forwards the request to that client. Handling of a datablock request from the request list to forwarding or handling the requests by the manager is common to all the algorithms.
Before the manager, client and server aspects of the implementation are discussed, there are a few constants objects and they are explained here. Since four algorithms are being implemented, the client implementation uses inheritance. That means there is a Client abstract class and the algorithm specific implementations are done in their respective client subclasses. An input parameter is used to decide which kind of client’s class has to be instantiated. An enumeration has been used to define the set of algorithms as constants. Similar to algorithms, enumeration type constants have been defined for experiment variable and Bloom filter type. The constants for experiment variable are Number of Clients and Cache size. As of Bloom filter type, the constants for it are Standard and IBF. Here Standard represents the standard Bloom filter and IBF represents Importance Aware Bloom filter. There is another class called Strings which contains string constants that do not belong to any particular category. They are mainly used while reading input parameter from the config file. All the above mentioned classes are present in the utility package and are mainly used by the simulator. Another important class present in this package that provides functionality to clients and manager is Hash. It has all the hash functions used to add and verify data memberships by client and manager objects. The hash functions used are Jenkins and Murmur, they provide 32-bit hash value and that is one of the main reasons to choose them. Hash class has a generateHashValues method which takes in the datablock value and returns an integer array of hash values.

5.1 Framework

The idea for our simulation framework is mainly inspired from Summary Cache algorithm. Like in Summary Cache our Proxy also maintains a Bloom filter. In the simulation framework there is only one proxy unlike in Summary Cache. All the communication from or to any client goes through the proxy. Proxy maintains a Bloom filter corresponding to every client to keep track of their cache content. The type of Bloom filter used depends upon the experiment. There are two types of Bloom filters that are used in the experiments. One is a standard Bloom filter and the other is an Importance Aware Bloom filter. In this section we are not referring to any one particular type of Bloom filter therefore the communication flow between clients, manager and server shown in figure 6 applies to both types of Bloom filter. Clients update their corresponding Bloom filters as soon as they receive a new datablock. A client keeps reading from its trace data and sends a datablock request to the manager if it couldn’t find the datablock in its own local cache. The manager then looks up in the array of Bloom filters to
determine which client’s cache has the requested datablock. If it finds such a client, it forwards the request to that particular client.

Since Bloom filters are used, the lookup process can result in false positives and false negatives. False negatives occur only when IBF is used because an IBF deletes random number of cells for every insertion. False positive means the manager wrongly indicates that a client has the requested datablock. We call a false positive a “miss”. When the manager’s lookup is correct and the client really has the requested datablock we call it a “hit”. Both hit and miss are used as evaluation parameters in our experiments.

![Communication flow between components](image)

**Fig. 6. Communication flow between components**

### 6. RESULTS
This section discusses about the predicted results for each algorithm and the results obtained from the experiments. All the results are represented in the form of graphs and each graph is explained.

#### 6.1 Predicted Results
This section discusses about the expected behavior of each evaluation parameter against each experimental variable. For all the algorithms this behavior will be similar, but position of the curves for each algorithm will be different because some algorithms perform better than the other. For any graph, except for the evaluation parameter and experimental variable all other affecting factors are assumed to be constant. In the below graphs, the hit count and disk access are actually measured in percentages of the total requests made. Here total requests are counted as total sum of requests made by all the clients. Hit count percentage and disk access percentage formulae are given below. The reason for plotting the graphs using percentages as opposed to the absolute values is the hit count will increase as the cache size or client count is increased. The graphs will look linear in that case and the hidden observations like stabilization of hit counts per total requests could not be made. For simplicity in the subsequent paragraphs of this section, hit count percentage and disk access percentage are called as hit count and disk access respectively. The following graphs are only the expected graphs and are predicted based on the research done during the initial phase of the project. They are only predictions and is used mainly to evaluate the outcome of the experiments.

\[
Hit \ Count \ % = \left( \frac{Hit \ Count}{Total \ Requests} \right) \times 100
\]

\[
Disk \ Access \ % = \left( \frac{Disk \ Access}{Total \ Requests} \right) \times 100
\]

In other words,

\[
Disk \ Access \ % = 100 - Hit \ Count \ %
\]
In figure 7, whose distribution looks similar to that of a logarithmic function, when cache size increases the hit count also increases. As we increase the cache size the hit count tends to stabilize. The reason for expected stability is that, as the cache size increases, for the same disk size, more number of datablocks are cached at the clients. As more number of datablocks are cached at clients, hits at clients also increase, but the rate at which hits increase decrease as cache size increases.

In figure 8, the distribution looks similar to part of $f(x) = x^{-1}$. The disk accesses are expected to reduce drastically and stabilize as cache size is increased. The reason for this behavior is, as cache size is increased, more number of datablocks are cached at the clients and the requests to the server will be reduced eventually.
Figures 9 and 10 show the expected behavior of hit count and disk access when client count is increased. They are similar to that of their respective graphs vs cache size. The reasoning is also similar to the cache size graphs. There were a few other graphs that were predicted, but haven’t been included here. For example, there was a prediction of hit count and disk access behavior when the bloom filter size is varied, but it only makes sense to use a bloom filter with optimum size. So, the experiments were conducted only with the optimum size.

6.2 Experimental Results
The graph plotted for Hit Count Vs Client Count experiment is shown in figure 1. It evaluates all the four algorithms against each other. Throughout this experiment, the client size is kept constant at 50. The trace data and initial cache warming for this experiment are random data. The size of randomly generated trace data for each client is equal to its cache size. The disk size is kept constant at 1000. The client count is varied in this experiment and corresponding hit counts for each algorithm is plotted. Initial client count is 10 and increased until 100 with an increment of 10 each time. Different color scheme has been chosen for each algorithm to be able to identify quickly. Color assigned to each algorithm is mentioned at the bottom of the graph. From the graph, it is evident that all the algorithms have their hit counts increasing with increase in client count. This is similar to the prediction made in figure 8. As for the performance of algorithms, RobinHood has more hit count when compared to the other three algorithms. It seems like RobinHood and NChance overlap, that is because of the difference in hit count is very less between the two and the fact that hit count represented here is in percentage, so the absolute difference is scaled down and looks insignificant. The hit count of NChance is significantly more than that of Greedy Forwarding for all the client counts. KSequence for K = 3 has been chosen to compare against the other algorithms. The reason for choosing K = 3 is that, for K = 1, KSequence is Greedy Forwarding and K = 3 looked like a reasonable value considering cache warming and trace data provided are randomly generated. Performance of KSequence (K = 3) is less than that of the other algorithms and this is the expected behavior. Evaluation of KSequence with different K values is discussed in later sections.

The graph plotted for Disk Access Vs. Client Count experiment is shown in figure 12. The cache warm-up and trace data are generated in the same way as that of the experiment represented in figure 10. The cache size and disk size are kept constant at 50 and 1000 respectively. This experiment evaluates all the four algorithms with respect to their disk accesses when client count is increased from 10 to 100 with an increment of 10 each time. RobinHood has low disk accesses when compared to all the other algorithms. RobinHood is represented by a yellow curve which is lower than all the other curves at all
points except for when client count = 10. The reason for that anomaly is random trace and cache data, but the important thing here is that RobinHood has the lowest disk accesses throughout rest of the graph. NChance has the second least number of disk accesses for all the client counts much lesser than Greedy Forwarding which it attempts to improve upon. Lastly, KSequence when K = 3 is observed to have more disk accesses for all the client values. All the algorithms performed exactly as predicted both in terms of decrement of disk accesses as client count increased and their disk access when compared to each other.

The graph in figure 13 shows the hit count Vs cache size experiment. Each algorithm, is executed with cache size increased from 10 to 100, incrementing 10 at a time. The client size and disk size are kept constant at 50 and 1000 respectively. Similar to the experiments represented in figures 11 and 12, the trace data generation and cache warm up is done randomly. Size of the trace data is equal to the cache size. In this experiment also RobinHood algorithm received more hits than the rest of the other three algorithms. Although there seem to be overlap between RobinHood and NChance, the absolute hit counts vary considerably. NChance clearly has more hit counts by a vast majority than Greedy Forwarding for every cache size. KSequence has the least number of hits as expected.
The graph in figure 14 represents the disk access vs cache size experiment comparing all the four algorithms. The settings of the experiment as same as that of experiment represented by figure 13. RobinHood has the least number of disk accesses for all the cache sizes. NChance has the second least number of disk accesses and performs far better than Greedy Forwarding. KSequence has the most disk accesses for any cache size. This outcome of the result is as expected.
Figure 15 represents an experiment conducted to evaluate the performance of standard Bloom filter and Importance Aware Bloom filter. We have compared the two types of Bloom filters on their false positive rate percentage. False positive rate percentage is measured as a percentage of false positives obtained for the total number of requests made by all the clients combined. Both the Bloom filters show an uneven increasing false positive percentage. But the IBF always has very less number of false positives percentage than that of the standard Bloom filter. The reason for this lies in Bloom filters’ construction. A standard Bloom filter only allows element insertion, it can quickly fill up and result in false positives. Whereas an IBF clears a random p number of cells for every element insertion and that prevents it from filling completely. Therefore, fewer false positives. Although IBF provides fewer false positives than standard Bloom filter, it introduces false negatives. Our framework does not support the measurement of false negatives. To measure false negatives, for every client request made, all the clients have to be checked to calculate false negatives given by the IBF and that would be an excessive overhead on the manager. However, the authors in [3] claim that there is an upper bound to false negatives obtained by IBF.

![Fig. 16. Standard Bloom Filter Vs IBF (Client Size)](image)

Figure 16 shows another experiment where a standard Bloom filter is compared to an Importance Aware Bloom filter when number of clients are increased. As in experiment of figure 15, this experiment also shows that false positives rate percentage of standard Bloom filter is more than that of Importance Aware Bloom filter for all client values. The reason for this behavior is also same as that of figure 15. A standard Bloom filter fills up quickly and gives false positives. Whereas an IBF clears a random p cells for every insertion operation.
Figure 17 shows an experiment where the number of empty cells present in Bloom filters after execution of NChance algorithm for different cache sizes. The cache sizes varied from 10 to 300, increasing by 10 at a time. From the graph we can see that rate of decrease of empty cells for Importance aware Bloom filter is less than that of a standard Bloom filter. This experiment is to prove that a standard Bloom filter fills quickly and gives more false positives. Whereas IBF tends to stabilize at a value of empty cells.

Figure 18. Hit Count Vs K-Value
Figure 18 the experiment evaluates KSequence algorithm. The cache size has been kept constant at 50 and disk size at 1000. The trace and cache data are generated randomly. Size of the trace data for each client is the size of its cache. For a set of clients, the K value was increased 1 to 10, incrementing by 1 at a time. The set of clients has also been increased from 10 to 100 incrementing by 10 at a time. The experiments represented by the next three figures are also conducted in the same environment. Each curve in the graph represents a set of clients. Two observations have been made in this experiment. The first, as the K value is increased from 1 to 10, the hit count for any given set has decreased. The second, rate of decrease varied for each set of clients. The drop of hit count for a set with fewer clients is more than that of the drop of hit count with more number of clients. The reason for the first observation is that as the number of clients increased the K value became more and more insignificant to the client set. Since each client tries to form a sequence of datablocks and forwards sequences in response to received client requests, these forwarded sequences replace existing datablocks from cache. Thus the global cache is being partially filled with unwanted datablock sequences. For the second observation: when the number of clients are less, the sequences formed by evicting the existing datablocks affected

![Fig. 19. Hit Count Vs K-Value](image)

Figure 19 represents the graph of hit counts of clients as K value is increased Vs hit count. Each curve in the graph has single K value. In other words, hit counts are calibrated keeping K value constant and increasing the client size in a curve. All the curves have same distribution, but their hit counts vary. The curve with lesser K value has more hit count. As the K value is increased, the hit counts decreased. The reason for that being, when the K value is 1, the algorithm acted like NChance [7]. As the K value is increased, the sequences began forming but the trace data does not have sequential blocks. This led the global cache to the decrement of datablocks in the global cache. Thus the reduced hit count.
Figure 20 shows the experiment in which sequential trace data is provided to a varied number of clients from 10 to 100 while keeping K constant. This experiment is similar to that of illustrated in figure 19. It is clear from the graph that for all the curves, hit counts have increased considerably when compared to the previous graph. The reason for this increase in hit count is the sequential trace data. Any sequence is being read from the trace, and that sequence is found in the global cache. If one block is present from a sequence, then all the blocks are present. Hence, more hits. Although it is expected that greater K value has more hit count, the observation shows the opposite. The reason for this is cache warmup and data generation. With higher K value, the number of random first block generations are fewer than that of lesser K value. Since the sequences are generated randomly, the probability of trace data present in the global cache is reduced. This is the main reason for decrease in hit count with increase in K value. The other reason is that, when a singlet sequence is forwarded, it evicts another sequence in a random client singlet sequence is forwarded to. This decreases total number of sequences in the global cache which affects overall hits.
Figure 21 shows the experiment in which sequential data is provided to varied number of client with number of clients kept constant and K value is increased. Each curve in the graph shows the behavior of a set of clients. This experiment is similar to experiment in figure 18 with only sequential trace data and caches. The first observation in this experiment is that, hit counts are directly proportional to number of clients. The reason for this is quite obvious. When more clients are present more sequences are present in the global cache hence more hits. Second observation: as the K value increases, the hits are reduced. The reason for this is similar to that explained for experiment in figure 20. Less sequences are present in the global cache so, the probability of having common sequences decrease. Eviction of sequences as a result of forwarding singlet sequences is another reason.

7. FUTURE WORK

We think that the treatment of KSequence like other cooperative caching algorithms at the local cache level is not appropriate. The performance of KSequence depends on the caching algorithms. Using LRU is not best solution because, it tends to evict blocks from sequences, forming incomplete block sequences. Although not all evictions result in incomplete sequences, but eviction of a singlet sequence high chance of doing so. As the forward client is being selected randomly in KSequence, the cache state of that client is unknown. It may result in evicting singlet sequences entirely or partially. To avoid this problem, we can incorporate a RobinHood like approach. Keep track of number of instances of each sequence present in the global cache. Pick the client to forward a singlet sequence to in such a way that the forward client makes space for the singlet sequence evicting a non singlet sequence which has many instances in the global cache. If no such non singlet sequence is found, then a random client can be picked and blocks which do not form a sequence can be evicted. Thus preserving sequences.

8. CONCLUSIONS

The conclusions are made based on the graphical analysis of the various experiments conducted. We have evaluated three existing algorithms and one proposed algorithm, all implemented using our simulation framework.

Most of the results obtained are as expected. First, the performance of algorithms came out as expected as well. RobinHood works best among all the four algorithms evaluated. As expected RobinHood performs better than NChance which in turn performs better than Greedy Forwarding. This is true for
both the Bloom filters that are evaluated. During the implementation stage, we observed that preventing the ripple effect that occurs from Singlet forwarding improves NChance’s performance.

Second, the performance of Importance Aware Bloom filter is better than that of a standard Bloom filter in terms of false positives and stabilization. But, we also noticed that IBF has very few more disk accesses than that of standard Bloom filter. This is due to the false negatives that IBF gives. While the main motto is to reduce disk accesses, IBF fails to improve performance at that. Also, when we consider the space aspect of the Bloom filters, we used cell size of 3 for IBF. That means IBF requires 3 times more memory space than standard Bloom filter. Also, the proxy uses a 32-bits for importance value of every data block. Not to mention computational overhead of importance function, which we do not have for our simulation, but that may not hold true for real world distributed file systems. There is also random p numbers generation which adds to the overhead. These are a few cons of using IBF because a Bloom filter is primarily used to minimize space and complexity.

Third, KSequence performs as expected with random trace data. But with sequential trace data, an improvement in the performance was expected. The reason could be the size of the trace data. With a large trace data it is possible that it shows better results.

We understand that the random function in Java has its limitations and its usage in creating synthetic trace data, and cache warmup is not ideal. An improvement over this can be the usage of actual trace data.

9. REFERENCES


