2011

Scalable cooperative caching algorithm based on bloom filters

Nodirjon Siddikov

Scalable Cooperative Caching Algorithm based on Bloom Filters

Nodirjon Siddikov

Email: nbs8816@rit.edu

M.S. in Computer Science

Department of Computer Science

B. Thomas Golisano College of Computing and Information Sciences

Rochester Institute of Technology

Rochester, New York

August, 2011

A Thesis submitted in Partial Fulfillment of the Requirements
for the Degree of Master of Science in Computer Science

COMMITEE MEMBERS:

_____

CHAIR: Hans-Peter Bischof DATE
Professor, Department of Computer Science

_____

READER: Minseok Kwon DATE
Associate Professor, Department of Computer Science

_____

OBSERVER: James Heliotis DATE
Professor, Department of Computer Science

Table of Contents

Table of Figures

1. Abstract

This thesis presents the design, implementation and evaluation of a novel cooperative caching algorithm based on the bloom filter data structure. The new algorithm uses a decentralized approach to resolve the problems that prevent the existing solutions from being scalable. The problems consist of an overloaded manager, a communication overhead among clients, and a memory overhead on the global cache. The new solution reduces the manager load and the communication overhead by distributing the global cache information among cooperating clients. Thus, the manager no longer maintains the global cache. Furthermore, the memory overhead is decreased due to a bloom filter data structure. The bloom filter saves memory space in the global cache and makes the new algorithm scalable. The correctness of the research hypothesis is verified by running experiments on the caching algorithms. The experiment results demonstrate that the new caching algorithm maintains a low block access time as existing algorithms. In addition, the new algorithm decreases the manager load by the factor of nine. Moreover, the communication overhead is reduced by nearly a factor of six as a result of distributing the global cache to clients. Finally, the results show a significant reduction in the memory overhead which also contributes to the scalability of the new algorithm.

2. Introduction

2.1. Cooperative caching

Distributed file systems and content distribution networks (CDN) use a cache to temporarily store the frequently or recently accessed data, so that future data requests can be served faster. While the performance of these systems is improved due to caching, their distributed architecture limits the cache efficiency. The solution to such limitation is to implement a cooperative caching mechanism, so that the clients can access each other's cache in case of a local cache miss. Consequently, the cache efficiency is improved. The cooperative caching is different from the traditional caching in terms of the number of caches. The cooperative caching involves using a global cache shared among clients besides the traditional local cache. The content of the global cache is made up of local caches and it is maintained by the manager. If a client experiences a local cache miss, it first contacts the manager to look up the global

cache. In case of the global cache hit, the clients cooperate with each other to serve the requested block. Otherwise, the client contacts the server to fetch the block from the disk. In non-cooperative caching systems, the global cache does not exist, so no cooperation happens among the clients. As a result, the request for the block skips the cooperation step and it directly goes to the server which may be an expensive process [1]. The cooperative caching usually utilizes an algorithm to control the local and global cache contents. Examples of cooperative caching algorithms are N-chance [7] and hint-based algorithms [8].

2.2. Research problem

The major problem related to existing cooperative caching algorithms is low scalability. These algorithms are not scalable when the number of clients or the local cache size is increased. One reason for such behavior is using a centralized approach to manage the global cache. In the centralized approach, the manager is responsible for looking up the global cache, forwarding the block request to an appropriate client and maintaining the global cache consistency. Because of so much responsibility, the manager eventually becomes overloaded when more clients are added to the system. Another reason for low scalability is the memory overhead on the manager. The manager has to maintain a large cache memory to store the global cache information. As the number of clients increase, the global cache size also increases gradually. This results in an increased memory overhead on the manager. The last problem is related to the communication overhead on existing algorithms. Every time a client updates its local cache, it has to report the changes to the manager so that it can update the global cache [7]. Therefore, increasing the number of clients results in an increased communication overhead between the clients and the manager. For example, the N-chance algorithm uses a single manager to maintain the consistency of the global cache, thus its performance is reduced as the number of clients increase [7]. On the other hand, the hint-based algorithm uses a hint that allows a client to make decisions based on local cache information. However, hints are not accurate most of the time, so this may cause a frequent global cache miss [8]. Hence, the clients still have to contact the manager for facts stored in the global cache. This increases the communication among the client and the manager which reduces the scalability of the algorithm.

2.3. Proposed solution

In order to address the research problem, this thesis develops a novel cooperative caching algorithm based on a bloom filter data structure. The bloom filter reduces the memory overhead on the algorithm by using

less memory for the global cache. Thus, the algorithm maintains its scalability when the number of clients or the local cache size increases. Moreover, the new algorithm uses the decentralized approach that distributes the global cache among clients. Consequently, the clients would have their own copy of the global cache and they no longer contact the manager for global cache lookup. This mechanism reduces the communication between the clients and the manager. Finally, the improved local cache replacement policy aims to increase the efficiency of the local and global caches by retaining the valuable blocks in the memory. The proposed solution is suitable for systems that work under client/server computer architecture. The specific contributions of the current thesis to the research field include devising a novel scalable cooperative caching algorithm, improving the local cache replacement policy and comparing the performances of the existing and proposed caching algorithms by using simulation software.

In order to evaluate the performance of the new algorithm, a trace-driven software simulator has been developed. The simulator verifies if the decentralized architecture of the new algorithm indeed reduces the memory and communication overheads. The simulator also examines if an improved cache replacement policy increases the efficiency of the local and global caches. Moreover, the experiment results demonstrate the effectiveness and correctness of the new algorithm. According to the results, the new algorithm performs better compared to the existing solutions when the number of clients or the local cache size increases. In particular, the bloom filter used the new algorithm contributes to the reduction of the memory overhead. Also, the communication overhead is decreased by factor of six and the manager load is reduced by factor of nine. Overall, the experiments results confirm that the new algorithm is more scalable compared to the existing algorithms.

2.4. Motivation

There are several issues associated with existing cooperative caching algorithms that triggered a motivation to do the current research. One of the interesting problems is an efficient use of a cache memory. The idea is to store as many blocks as possible in the cache without increasing its size because the cache that contains more blocks would have higher hit rate. The existing algorithms reach this objective at the cost of large global cache memory. However, the more clients are connected to the cooperative caching system, the more memory space is consumed by the global cache. One way to resolve this issue may be to distribute the global cache among the clients. But this solution affects the local cache hit rate of clients because global cache information requires more space in the client’s cache memory. This would leave a little space for client’s local cache information. However, applying data compression techniques could reduce the memory waste in the client’s cache. So, saving a space in cache

memory is one of the motivations to design the new caching algorithm. Another interesting problem associated with the exiting algorithms is their scalability. As the number of clients or local cache size increases, the manager becomes overloaded and the communication overhead between the clients and the manager increases. This would potentially reduce the algorithm's performance. Thus, this research aims to develop a scalable algorithm so that its performance and effectiveness is maintained. The final motivation for research is to explore the relationship between cryptography and caching algorithms. For example, the bloom filter data structure establishes this relationship because it uses cryptographic hash functions and can be applied to caching algorithms [2]. Moreover, bloom filters and hash functions are relatively new in the current research area, thus it is interesting to see how they help solving the scalability problem in existing algorithms.

2.5. Organization of paper

Before describing the research contributions, the findings of literature review are presented. For example, there are several background concepts that have to be defined and formalized, such as cryptographic hash functions, data compression methods and cache replacement policies, before the new algorithm is introduced. Therefore, the remainder of this report is organized as follows: Section 3 outlines the research hypothesis. Section 4 describes the required background knowledge and concepts. Section 5 elaborates more into existing caching algorithms that address the research problem. The new algorithm is introduced in Section 6 which is followed by Section 7 that describes the experiments and the metrics used to test the algorithms. Section 8 outlines the list of deliverables in current research. The limitations of this thesis and the future work in the research field are discussed in Sections 9 and 10, respectively. Finally, Section 11 summarizes the thesis report by presenting the conclusions drawn from the experiment results.

3. Research process

3.1. Research questions

Current research investigates several specific questions: is it possible to make cooperative caching algorithm more scalable by reducing its global cache size and decreasing the load on the manager? Which

way of reducing the global cache size is better: using data compression techniques or applying a probabilistic data structure such as bloom filter? Where else in the cooperative caching system the bloom filter can be used? For example, the bloom filter can be used in a local cache besides the global cache to further reduce client's cache size. Thus, the question of interest is in which case the bloom filter can save more memory? The scalability problem being investigated has an implementation in commercial cooperative caching solutions used by CDN providers such as Akamai. Hence, finding answers to these practical research questions is valuable for both the academia and industry.

3.2. Hypothesis

The scalability of cooperative caching algorithms can be improved by reducing the memory and communication overheads, and also by decentralizing the global cache management. The memory overhead can be reduced by replacing an accurate state of the global cache by an approximate state using probabilistic data structures. A probabilistic data structure, such as bloom filter, can be used for global cache information because it is faster, more compact and consumes less memory than deterministic data structures. Also, the manager's load can be reduced by distributing its responsibility to clients, so that the manager can handle more clients in the system.

3.3. Research goal

The goal of this research is to verify the correctness of the proposed hypothesis by using several supporting evidences such as formal analysis, modeling and experiments. For example, the formal analysis of the bloom filter demonstrates its efficiency and advantage over hash table data structure. Besides analysis, the detailed architecture of new cooperative caching algorithm explains the reasoning behind its design and components. On the other hand, the mathematical model of the bloom filter elaborates more about its internal structure. For example, the mathematical model shows that combination of an array and cryptographic hash function can produce an acceptable solution for the research problem. Finally, the experiments aim to push the boundaries of the research hypothesis and find any counterexamples that disprove it. The expected phenomenon that should be observed in the case of a valid hypothesis is a trend: as more clients join the cooperative caching system, the communication and memory overheads grow linearly but not exponentially. Therefore, such expected behavior confirms the scalability of new algorithm.

3.4. Research scope and limitations

This research focuses on important cache management policies such as cache organization and lookup, block removal and relocation. Particular attention is paid to the application of a probabilistic data structure, such as a bloom filter, in cooperative caching to produce a scalable algorithm. The research on bloom filters unveiled that they consume less memory than traditional data structures such as hash tables or arrays [19]. For example, it takes less memory if the global cache information is store in a bloom filter rather than in a hash table. However, the convenience of using the probabilistic data structure comes with a cost: a bloom filter only stores an approximate state of the global cache. Moreover, the approximate state produces false positive and negative hits. Consequently, the global cache hit ratio may be reduced which contributes to the increased block access time. Also, an improper construction of bloom filter also produces an unstable data structure that increases the false cache hit and miss rates. Thus, one of the challenges in designing the new algorithm is to ensure that the bloom filter is constructed with parameters that use optimal values. As long as the bloom filter uses optimal values for its parameters, it produces less false positive and negative hits. Another limitation of the research is using a block based granularity in the new algorithm. A block based granularity is much finer than file based granularity because a file is usually composed of one or more blocks. Such finer granularity makes it easier for the new algorithm to perform operations such as cache lookup. However, in real world applications, a file based granularity is used. Therefore, the new algorithm does not reflect the true nature of a file based system. Finally, the experiments used synthetic traces to evaluate the performance of algorithms. However, these traces do not replicate the real world user generated block access pattern. Using traces obtained from distributed file systems or CDNs would produce more accurate and reliable experiment results.

4. Background review

4.1. Cache memory

A cache memory plays an important role in computer engineering because it transparently stores the data so that future data requests can be served faster [12]. The data stored in the cache may either be a previously computed value or the block that has been requested before. The cache memory also improves the performance of secondary storage devices, such as a hard drive, by saving the copies of frequently or

recently accessed blocks. For example, on most computer systems, accessing the disk is relatively slow compared to the speed of the main memory. Therefore, to accelerate the repeated access to the blocks, the recently or frequently used blocks are cached in the main memory or some other sort of faster memory [17]. In a client/server environment, caching is used to reduce the request latency, service time (i.e. round trip time) and the network traffic.

The cache memory is a part of the computer memory hierarchy ordered from the fastest memory at the top to the slowest one at the bottom (Figure 1). The main role of the cache is to filter out the access to slower layers of the memory hierarchy. Thus, a block request that hits the cache avoids accessing the slower memory. For example, if the requested block is in the cache (i.e. a cache hit), it is served directly from the cache memory, which is relatively faster. Otherwise (i.e. a cache miss), the block has to be fetched from the slower memory layer such as a disk. Therefore, the performance of the system will be improved if more blocks are read from the cache [12]. The cache size is usually kept small to make the cache cost efficient. Moreover, the cache also demonstrates an access pattern also known as locality of reference. According to this concept, only a small number of blocks are accessed at any given point in time. The cache usually exhibits two types of locality: temporal and spatial. Temporal locality states that if a particular block is referenced, then it is likely that same block will be referenced again in the near future. Spatial locality dictates that if a particular block is referenced at a particular time, then it is likely that nearby blocks will be referenced soon [15]. Most cooperative caching algorithms discussed in this research utilize the locality of reference concept.

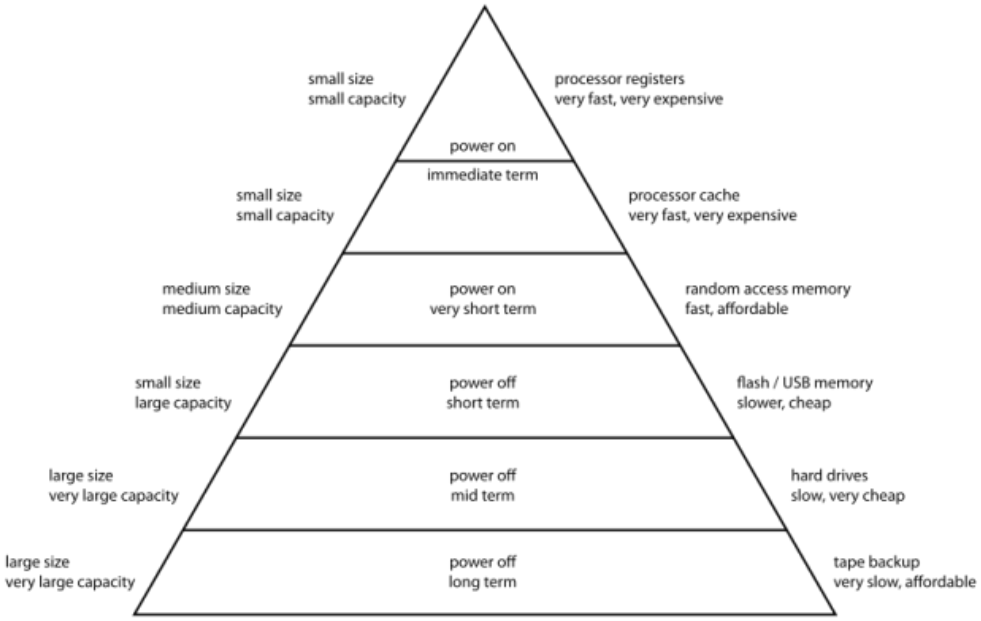

Figure 1: *A computer memory hierarchy [13]. In one extreme, the cache size is smaller but its performance is faster, however it is otherwise in another extreme.*

4.2. Elements of cache design

When designing a new cooperative caching algorithm, the following elements are used to increase its effectiveness: the cache size, a mapping function, a cache replacement policy, a write policy and the block size [21].

4.2.1. Cache size

The cache size determines the capacity of cache memory. Increasing the cache size decreases the cache miss rate. For example, the larger is the capacity of the cache, the more blocks it can accommodate; consequently, the cache hit rate will be increased. However, the benefit of increasing the cache size is limited up to a certain threshold. If the cache size passes this threshold, no more improvement is seen in the efficiency of the cache. The large cache size causes no block request to be forwarded to the server. Moreover, the large cache involves using more gates to access the block. Thus, the size of cache has to be kept small to maintain its efficiency. The cache size is also limited due to a space available in hardware chip. However, there is no optimal size for the cache capacity, because the workload over cache memory is different in various applications [21]. In this research, the cache size varied from small to large during experiments to measure the new algorithm's scalability.

4.2.2. Mapping function

The mapping function defines which file block is hosted by which cache block. Caching algorithms usually use two types of mapping functions: direct and associative.

Direct: this function maps one file block into one available cache block. Because the number of file blocks is more than the number of cache blocks, many file blocks can be mapped into the same cache block. Even though this approach is the simplest, it has a major disadvantage: the disk block is always mapped into the same cache block and this mapping remains permanently. For example, if two file blocks are mapped into the same cache block and they are accessed continuously, these two file blocks will constantly be swapped. This would cause a cache miss and increase the block access time [20].

Associative: this function overcomes the limitation of the direct mapping function and allows any file block to be mapped into any cache block. In this case, the cache block maintains a special tag to represent the file block's index. The disadvantage of this method is associated with the search operation. Searching

for a specific file block involves visiting all cache blocks in order to find one that is mapped into the requested file block [20].

Mapping function	Hit ratio	Search performance
Direct	good	best
Associative	best	moderate

Figure 2: *Comparison of mapping functions [21].*

Figure 2 shows the comparison of the mapping functions. The new algorithm presented in this research implements the associative mapping function because of its better hit ratio.

4.2.3. Cache replacement policies

A block lifetime is an important factor in designing a cache replacement policy. Once the lifetime ends, the block becomes expired or the least valuable. Due to the limited physical size of the cache, the expired or the least valuable block is removed and replaced with new one. A cache usually implements a replacement policy to deal with expired blocks. According to this policy, an expired block remains in the cache if there is enough space; otherwise it is removed immediately. In case of direct mapping, there is only one possible cache block for any file block [21]. Thus, no replacement policy has to be enforced. However, the new algorithm uses associative mapping for local and global caches; therefore, all file blocks are candidates for replacement.

Cache replacement policies are usually grouped into online and offline policies. An online policy uses the sequence of block requests as an input and this sequence is not known ahead of time. On the other hand, an offline policy has the information about future block request and it retains the blocks in the cache to increase the hit rate. Therefore, offline policies usually have better cache hit rate compared to online policies. However, offline policies are not implemented in practice, but instead they are used as theoretical best case in the experiments. Examples of online cache replacement policies are LRU (least recently used), LFU (least frequently used) and FIFO (first in first out). Belady's algorithm is an example of an offline cache replacement policy. Most of the cooperative caching algorithms use a specific cache replacement policy in local and global caches.

FIFO: it is a very simple cache replacement policy. In this policy, the blocks are added to the cache in the order they are accessed. All cached blocks are stored in a queue data structure and their order is not changed. When the cache becomes full, the blocks are removed from the queue in the order they were

added. This policy can be implemented very easily and it works fast. In a simplified case, it only takes an array with an index to implement it. However, this simplicity comes with a price: FIFO policy is not smart and it removes blocks regardless of their access frequency [17].

LRU: like FIFO, this policy caches the blocks in the order they are accessed. But unlike FIFO, when the cache becomes full, the policy discards the least recently used block to make space for the new one. The LRU policy keeps track of each block's access time to determine the least recently accessed block. This policy can be implemented using a linked list so that the most recently accessed block is moved to the head of the list. On the other hand, the least recently accessed block ends up being at the tail of the list from where it is removed. This policy is simple and fast. It has an advantage over FIFO on adapting to the access pattern by keeping recently used blocks in cache for a longer time [17]. The proposed cooperative caching algorithm uses LRU to manage the local cache content.

LFU: this policy retains the most frequently accessed blocks in the cache. Access frequency is maintained by attaching a counter to each block. Every time the block is accessed, the counter is incremented. When the cache becomes full, the block with the lowest access frequency is removed. If there is a tie between counters of two blocks, LRU algorithm breaks the tie. Due to the bookkeeping of block access frequency, LFU experiences extra overhead. Moreover, LFU does not adapt quickly to the block access patterns and it takes time until these access patterns are recognized [17].

4.2.4. Write policy

Write policy ensures that if the cached block's content is modified, the update is reflected in the server. Two main writing policy methods are "write-through" and "write-back" [21]. The scope of the current research excludes the consideration of the writing policy; therefore, it is not discussed further in the paper. However, the write policy is considered as a possible extension to the new algorithm and it is included as a future work.

4.2.5. Block size

The block size is an important factor when designing the new caching algorithm. Typically, when the block is fetched from the server and placed into the cache, the neighbor blocks in close proximity are also cached. At first, this approach may result in higher hit ratio due to a spatial locality. However, if the block size increases, cache hit ratio gradually declines. The reason for such behavior is that larger blocks take

more space in the cache, so the cache can accommodate less blocks. Consequently, the cache hit rate may be reduced over time [21]. This thesis assumes that the cache block size is the sum of tag size and file block size. Moreover, the cache block can host only one file block.

4.3. Architecture of cooperative caching system

A typical cooperative caching system usually implements the central global cache which is made up of client local caches. The global cache is a layer in the computer memory hierarchy and it is positioned between a local cache and the disk [8]. Some global cache implementations store merely the information about local caches rather than the actual cache content. The cache replacement policy used in the global cache is usually different from that of traditional cache. The content of the global cache is managed by a cooperative caching algorithm which is quite complicated. For example, the client cannot simply remove its least valuable block, but instead the block has to be compared against all other expired blocks in the cooperative caching system. Once the least valuable block among all local caches is found, the block becomes a candidate for removal from the global cache [8].

The architecture of cooperative caching system usually involves a manager, the clients and a server. The manager maintains the global cache; the clients receive block read requests from the users and the server hosts the blocks. When the clients access the block in the server disk, the manager usually coordinates this process. The coordination provided by the manager may also include global cache lookup and block relocation operations. The extent of coordination is different among cooperative caching algorithms and it is considered a major distinction between them [8]. Moreover, the clients have to cooperate with each other when serving a block to the user. Cooperation entails clients to access each other's local cache in case of cache miss.

Cooperative caching systems are usually designed using a centralized approach. In this approach, if a client receives a read request for a block but experiences cache miss, the request is forwarded to the manager to lookup the global cache. The manager obviously knows which client's cache has the block. If there is a global cache hit, the manager redirects the request to the appropriate client. Otherwise, the manager fetches the block from the server which may be a slow and a very expensive process. Figure 3 illustrates the clients that cooperate with each other to serve the block. According to the figure, the client 1 receives a request for block 'abc' but it experiences the local cache miss. So, it looks up the global cache at the manager. The lookup operation reveals that client 4 has the block 'abc'. Then, the client 1 contacts the client 4 through manager asking for block 'abc'. Eventually, the 'abc' block is served from client 4, rather than from the server.

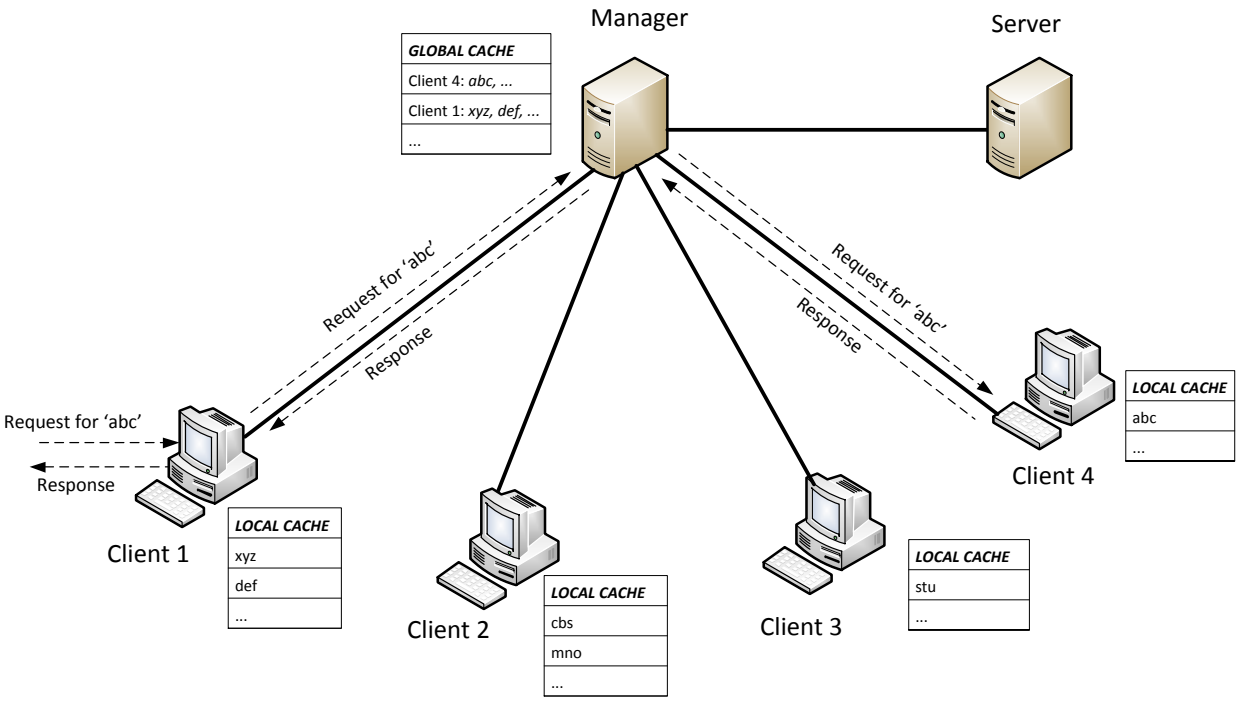

Figure 3: *A centralized cooperative caching system. In this system, a single manager is responsible to maintain the global cache.*

Figure 4 shows the layers of memory hierarchy used by the cooperative caching algorithm:

- ***Step 1:*** the client receives a block read request from a user;
- ***Step 2:*** the client looks up its local cache for a block;
- ***Step 3:*** if there is a cache hit, the client reads the block from its local cache;
- ***Step 4:*** if there is a cache miss, the client contacts the manager to look up the global cache;
- ***Step 5:*** if there is a global cache hit, the block is fetched from appropriate client's local cache;
- ***Step 6:*** otherwise (a global cache miss), the request is forwarded to the server;
- ***Step 7:*** the block is fetched from server and returned to the client;
- ***Step 8:*** the client saves the copy of block in its local cache and returns the original block to the user.

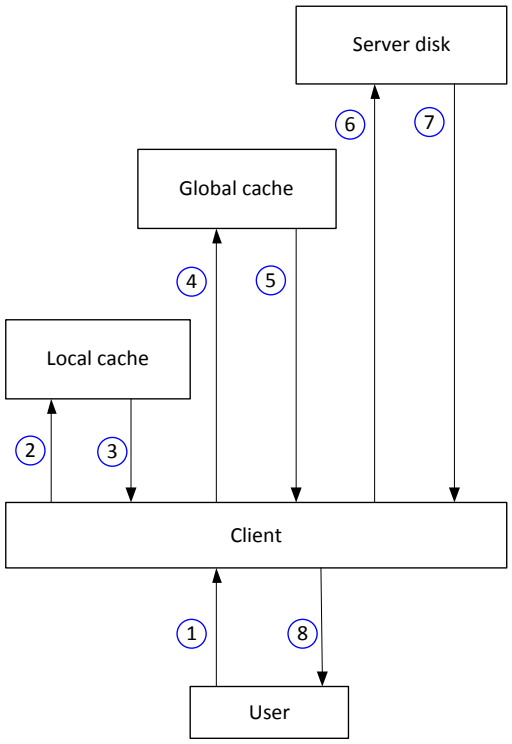

Figure 4: *Layers of memory hierarchy used by the cooperative caching algorithm.*

5. Existing algorithms

5.1. N-chance algorithm

The N-chance algorithm is one of the pioneers in the field of cooperative caching. It uses a centralized management approach to maintain its global cache. In this algorithm, the clients are responsible for maintaining their local cache; the global cache is maintained by the manager. Moreover, the global cache is shared among clients so that they can cooperate with each other when serving the blocks to the user. A local cache is managed by a simple LRU cache replacement policy, but the global cache is coordinated using a more sophisticated policy. In this policy, the clients either discard or forward the least favorable block based on the number of copies of the block in the system. If the cooperative caching system contains only one copy of a specific block, that block is called a singlet, otherwise it is considered as a non-singlet. If the least valuable block is a singlet, that block is forwarded to a randomly selected client to further retain it in the system; otherwise, the block is discarded immediately. Each singlet block is given a

recirculation count *N* such that every time the block is forwarded, the count is decremented. Therefore, the disk block is given *N*-chance to stay in the cooperative caching system. When the count reaches zero, the block is discarded regardless of its singlet status [7].

The central manager component plays an important role in the N-chance algorithm. The manager is responsible for global cache maintenance and lookup, and forwarding the block requests to the clients and the server. Every time client caches or removes a block, it reports these changes to the manager. The manager uses the information about changes to maintain the consistency of the global cache [7]. While the N-chance is a decent cooperative caching algorithm, its scalability is reduced as the number of clients or local cache size increases. For example, increasing the number of clients results in a sharp increase in the communication overhead among the clients. Moreover, the N-chance algorithm consumes more memory space for global cache when the number of clients increases.

5.2. Hint-based algorithm

In N-chance algorithm, the central manager is solely responsible for the maintenance of the global cache that keeps the facts about the local cache contents. While these facts maintain the accurate state of the global cache, keeping them up-to-date increases overhead on the manager. In order to address this problem, the hint-based cooperative caching algorithm aims to relax the centralized control of global cache. In particular, the algorithm distributes the portions of the global cache information among clients as hints, so that clients can make local caching decisions such as forwarding the block requests and relocating the blocks. On the other hand, the manager retains the control of the global cache, but it incurs much less overhead from clients. Therefore, in the hint-based algorithm, the global cache is maintained by both clients and the manager [8].

The hint-based algorithm uses the global cache to maintain the information about the location of a master block. The master block locations are stored as facts or hints. The facts are stored in the manager and the hints are stored in the client. When the block is fetched from the server for the first time and cached by a client, it is designated as a master block [8]. The copy of the master block is called non-master block. The hints reduce the client's dependence on the manager when performing such operations as request forwarding or block relocation. However, the information provided by the hints is not accurate at all times. The reason is that the hints are local to the client and they do not reflect the changes taking place in the caches of other clients. For example, if a hint tells the client about the location of a particular master block, it is not guaranteed that the block is present at the remote client's cache. The chances are that the remote block might have been forwarded to another client or it has been discarded entirely. Also, the

information about this change might not have been reflected in the client's hint. Thus, the hint gives client only a clue about the probable location of a block in the cooperative caching system. On the other hand, a hint prevents the client from frequently contacting the manager [8].

The cooperative caching operation in the hint-based algorithm is slightly different than in the N-chance algorithm. In hint-based algorithm, if a client experiences a cache miss as a result of an inaccurate hint, it contacts the manager for a fact. Once the client obtains the fact, it updates its appropriate hint with new master block location information. Moreover, when the remote client discards or relocates the master block, it updates the manager's global cache with a new location of the master block. On the other hand, hints improve the performance of the caching algorithm and make it slightly scalable. This slight scalability is contributed by the clients that partially share the responsibility of global cache maintenance. Consequently, sharing of management among the clients reduces the overhead on the manager. Moreover, managing hints in each client is less costly than managing facts, because the accuracy of hints needs not to be maintained at all times [8]. While the hint-based algorithm promises better global cache management, it introduces some problems that prevent the algorithm from being fully scalable. For example, the block removal process may replace the master block with a useless non-master block. Consequently, the local hints in other clients become obsolete and they will be required to contact the manager to obtain the fact. This results in an increased communication overhead as the number of clients increases. Also, it may take a while until hints are updated, thus a client is likely to make an inaccurate local decision based on the obsolete hint. Such operation is unfavorable since inaccurate decisions increase the cache miss rate [11].

6. New algorithm

6.1. Problems with existing algorithms

There are several problems associated with existing cooperative caching algorithms. The first problem is related to the scalability of the algorithms. Current algorithms do not scale well when more clients are connected to the cooperative caching system. As the number of clients increase, the number of messages exchanged among clients grows exponentially and the manager becomes overloaded. For example, in the N-chance algorithm, the singlet block is relocated to the remote client through the manager. When the number of clients increases, such block relocation procedures happen more often. Consequently, the manager has to do more work on the client's behalf to relocate the singlet block to the appropriate client.

The second problem is associated with a memory overhead on the global cache. Existing algorithms consume large memory space for the global cache. As more clients are added to the system, the manager has to allocate more memory space for the global cache. While some of the existing algorithms distribute the global cache information among the clients, they still suffer from non-scalability. In such distribution approach, the clients are responsible for maintaining the ever growing global cache information which occupies a substantial space in the local cache. Leaving less space for the locally cached blocks increases the client's cache miss rate. Overall, the communication and memory overheads make the existing algorithms unworkable when the number of clients or local cache size increases.

The current thesis aims to develop the scalable cooperative caching algorithm that addresses the problems related to the existing solutions. One of the goals of the new algorithm is to offer a scalable solution by reducing the global cache size for two reasons. First, the global cache occupies the large fraction of client's local cache and this reduces the cache hit rate. Second, the global cache is not used as often as a local cache. The global cache is contacted only if a client experiences a local cache miss. Thus, the global cache does not have to be kept up-to-date at all times. Besides offering a scalable solution, the new algorithm also resolves the inefficient LRU policy used in the clients' local cache. For example, during the block removal process, the LRU does not take into account the value of the block. Consequently, the valuable master block or the singlet may be removed from the cache. As a result, a local cache may become occupied with the less valuable non-master or non-singlet block. So, the new algorithm implements custom local cache replacement policy that retains the most valuable blocks in the cache for a longer time to increase the cache hit rate.

6.2. Cache summary

In order to reduce the global cache size, this research uses a cache summary. The cache summary represents the list of the blocks cached in the memory, but the contents of the blocks are not included in the summary. Therefore, the size of the cache summary is usually less than the actual cache. Applying the cache summary concept into the global cache may help the existing algorithms to be scalable. Moreover, the manager would have an ample free space in its global cache to store more local cache information. The cache summary is also used in decentralizing the global cache management. In this case, the global cache content is stored as a cache summary in each client. The cache summary does not have to be accurate or up-to-date at all times. For example, it does not have to be updated every time the new block is cached, but instead the updates can happen at regular time intervals [2]. Therefore, using the cache summary not only reduces the global cache size, but it also results in decreased communication overhead.

A decentralized algorithm that uses cache summary resolves the communication overhead problem incurred by the exiting solutions [2]. In typical client/server architecture, the communication overhead is a set of messages exchanged among the clients and the server. In case of decentralized algorithm, the overhead is calculated as the frequency of global cache update and block relocation operations. The frequency of updates is reduced either by delaying the update or by running the update in regular time intervals. The first method entails the delaying of the global cache update until the amount of newly cached blocks reaches a certain threshold. The second method simply updates the global cache on a regular time interval [2]. The current research applies the second method to the new algorithm because the content of the global cache tends to change frequently.

The cache summary is usually stored in a primary memory rather than a disk for two reasons. First, the primary memory is much faster than the disk. Second, the disk arm creates input/output (I/O) bottleneck when reading or writing the data to the disk. However, this bottleneck can be avoided by using a recent secondary storage technology such as a flash memory or a solid state drive [10]. In order to store the cache summary in the primary memory, it first needs to be created. There are two popular methods used to create a cache summary: data compression and data approximation.

6.2.1. Data compression

The data compression method compresses each block in order to reduce the overall cache size. However, block's ID is left uncompressed because it will be used as an index in the cache lookup operations. The rest of the block's attributes, such as its size and last access timestamp, are compressed. The data compression operation can be performed efficiently by using third-party software libraries. These libraries simplify the implementation of the new caching algorithm [10]. However, there are several problems associated with data compression. First, the compression operation may not be effective for small size blocks. The small size blocks do not use much memory, so it may not be necessary to compress them at all. Second, the compressed blocks have to store metadata information in order to be decompressed. The experiments conducted over *zlib* software library shows that compressing the small sized block actually increases its final size due to its metadata overhead [10]. This problem can be resolved by compressing a group of blocks together. However, if one of the blocks needs to be accessed in the group, the entire group of blocks has to be decompressed. Alternatively, a special compression algorithm may need to be developed that works efficiently with small size blocks. Third, the compression/decompression operations are computation intensive. This places an additional computational overhead to each block read request which may overload the cooperative caching system [10].

6.2.2. Data approximation

The data approximation method entails keeping an approximate state of the cache. One way to approximate the cache content is to use a probabilistic data structure called a bloom filter. The bloom filter is an array that only requires a block's ID to insert the block into the cache [18]. It also determines the presence of the block in the cache with high probability. However, this probabilistic data structure may cause the following types of errors: false caches miss (i.e. false negative) and false cache hit (i.e. false positive). The false cache miss happens when the particular block is stored in the cache, but the bloom filter says otherwise. In this case, the block request is not sent to the remote client that actually possesses the block. On the other hand, the false cache hit occurs when the block is not stored in the client's cache, but the bloom filter says otherwise. Consequently, the request is redirected to the remote client, but no block is received in return. Even though these errors affect the cache hit ratio, using a bloom filter does not influence to the correct operation of the new caching algorithm. For example, a false cache hit does not result in a wrong block being served [2]. The chance of getting false positives and negatives depends on the values of the parameters used to create a bloom filter. There are several advantages using bloom filter in the global cache. One of them is that the size of the bloom filter is fixed so that it does not use too much memory space. Also, the probability of getting false positives and negatives can be controlled by proper choice of values for bloom filter parameters. On the other hand, the bloom filter can only be used for cache lookup operation, and it cannot be used directly to store the block's other attributes such as its size [10]. Moreover, data approximation method has a tradeoff between the memory space and the data accuracy. For example, if the size of the cache memory reduces, the level of data approximation increases.

6.3. Bloom filter

6.3.1. Mathematical description

A bloom filter is a probabilistic data structure that uses cryptographic hash functions. From a mathematical perspective, the bloom filter is a way of representing a set $A = \{a_1, a_2 \ldots a_n\}$ of n elements using an m-bit array and a set of cryptographic hash functions $H = \{h_1, h_2 \ldots h_k\}$. Initially, all values in the array are set to zero. Then, each of the hash functions, $h_1, h_2 \ldots h_k$, receives an element as an input and produces a result within range $\{1, 2 \ldots m\}$. When inserting an element a into a set A, the values at the positions $h_1(a)$, $h_2(a)$, … , $h_k(a)$ of an array are set to one. A deletion operation resets the values at the same positions back to zero. In order to check if an element is in the set (i.e. $a \in A$), the same procedure as in the insertion operation is repeated and the results of hash functions are compared against the array

values. If any of value is zero, then $a \notin A$. Otherwise, it is speculated that $a \in A$ with certain probability. For example, a particular bit might be set to one multiple times because of inserting a different element into the same bloom filter. Thus, the bloom filter may claim that $a \in A$, but it may not be the case with certain probability [2].

The bloom filter is usually created using the three important parameters: the bit-array size, the number of expected elements and the number of hash functions. The bit-array size, designated as *m*, defines the capacity of the array used in the bloom filter. Once the array is created, its size is not changed throughout the life of the bloom filter. The number of expected elements in the array, designated as *n*, defines the maximum number of blocks to be inserted into the bloom filter. Finally, the number of hash functions, denoted as *k*, defines the quantity of the functions to be applied for each of the elements. Improper construction of the bloom filter may result in conditions known as a false positive and a false negative. For example, two elements *a* and *b* are inserted into the bloom filter using hash functions $h_i(a)$ and $h_j(b)$. If these hash functions have the same output, the elements *a* and *b* are mapped into the same bit position in the array [16]. When the element *a* is removed from the bloom filter, the value at bit position $h_i(a)$ is reset to zero. This creates a potential false positive condition because the bloom filter now claims that the element *b* still exists. However, removing the element *a* also flipped the value at bit-position $h_j(b)$ to zero causing the element *b* not to be in the bloom filter. Another example is that if $n > m$, the bloom filter may become unstable and produce more false positives and negatives. Therefore, the values of the parameters *m, n* and *k* should be chosen carefully when creating a bloom filter so that the probability of false positives and negatives are acceptable [16]. The idea behind the bloom filter is presented in Figure 5.

6.3.2. Tradeoff between bloom filter parameters

There is a tradeoff between a bloom filter parameters *m* and *n*, and its probability of false positives. In order to measure this tradeoff, an equation between the parameters *m, n* and *k* has to be derived. The probability that a certain bit in the bloom filter is not set to one by a hash function while inserting an element is $1 - \frac{1}{m}$. The probability that the bit is not set to one by any of the hash functions is $\left(1 - \frac{1}{m}\right)^k$. If *n* elements were inserted, the probability that a certain bit is still zero is $\left(1 - \frac{1}{m}\right)^{kn}$; therefore, the probability that the same bit is one is $1 - \left(1 - \frac{1}{m}\right)^{kn}$ [19]. Now, let us assume that each of the *k* array positions in the bloom filter is equal to one with probability as above. The probability that all of *k* array

positions are equal to one, or a particular element is in the set (i.e. a false positive), would be $\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k$.

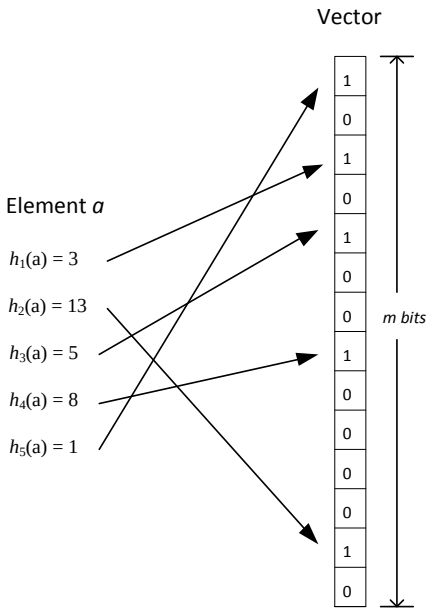

Figure 5: *The bloom filter data structure with hash functions [2].*

The right hand side of the equation is simplified to $k = ln2 * m/n$ which becomes as $\left(\frac{1}{2}\right)^k = (0.6185)^{m/n}$ and it is simplified to the expression $2^{-k} \approx 0.6185^{m/n}$. This expression shows the correlation between the three important bloom filter variables *m*, *n* and *k*. Calculating an optimal value of *k* and substituting it in the above formula gives the formula $p = \left(1 - e^{-\left(\frac{m}{n}ln2\right)\frac{n}{m}}\right)^{\left(\frac{m}{n}ln2\right)}$ which can be simplified to $lnp = -\frac{m}{n}(ln2)^2$. Solving the equation by *m* results in the following equation [19]:

$$m = -\frac{n\ lnp}{(ln2)^2}$$

Plotting the above equation into a coordinate plane gives insight into the relationship between the bloom filter parameters *m* and *n*, and its false positive probability (Figure 6). The plotted graph reveals that in order to keep the value of a false positive probability fixed, *m* (the length of an array) and *n* (the expected number of elements) have to grow linearly.

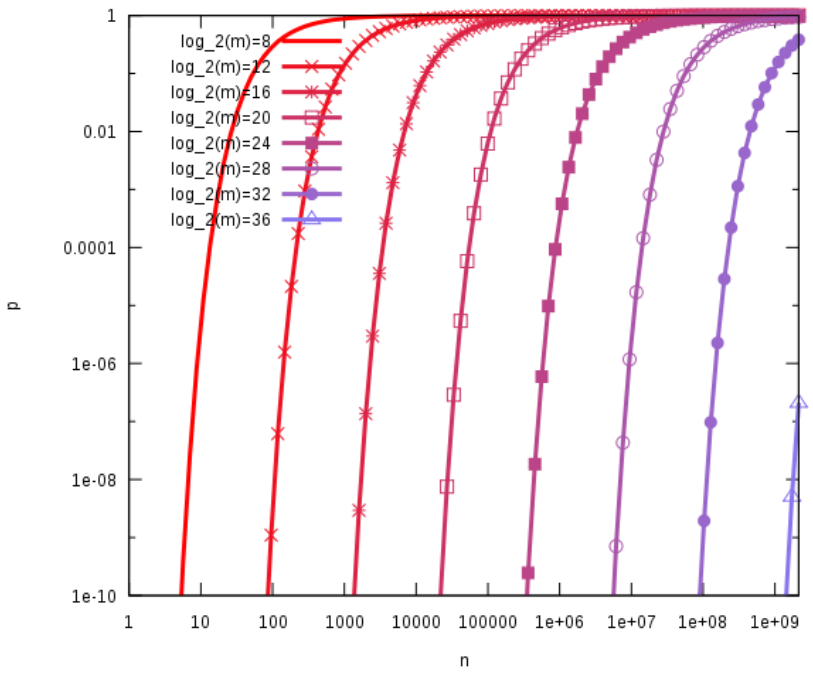

Figure 6: *The false positive probability p as a function of the number of elements n in the bloom filter and its size m. An optimal number of hash functions* $\boldsymbol{k = (m/n) * ln2}$ *have been assumed [19].*

Figure 7 shows the probability of a false positive as a function of the number of bits allocated for each element (i.e. *m/n*). The upper curve is produced in case of four hash functions. The lower curve produced in case of the optimum number of hash functions. It can be concluded from Figure 7 that the bloom filter consumes a little space per element with a slight risk of a false positive. For example, for an array ten times larger than the number of cached blocks, the probability of a false positive is 1.2% with four hash functions, and 0.9% for the optimum case of five hash functions. The probability of false positives can be easily decreased by allocating more memory [2]. In other words, to maintain a fixed false positive probability, *m* (the length of an array) has to grow linearly along with *n* (the expected number of elements to be inserted into a bloom filter).

6.3.3. Comparison of a bloom filter and a hash table

This section compares the two data structures: a hash table and a bloom filter. The hash table is often used in caching algorithms where each block is stored as an entry in the table. One of the problems associated with the hash table is its tradeoff between the memory usage and the hash collision. In order to reduce the likelihood of the hash collision, more memory space needs to be allocated in the hash table. For example, the rate of hash collision can be reduced by using a load factor value or a self-balancing tree. The value of the load factor defines how the hash table entries are far apart from each other. If the entries are further

located from each other, there is less chance of hash collision. However, the load factor value has to be selected carefully because when this value is close to zero, the amount of unused entries in the hash table increases [14]. Thus, the memory is wasted due to using the empty entries in the hash table. The other method involves using a self-balancing tree for collision resolution. Even though the self-balancing tree reduces the time complexity of common hash table operations, such as insertion, deletion and lookup, from $O(n)$ to $O(log\ n)$, it still consumes significantly more memory. Thus, the collision resolution strategies, such as a load factor value and a self-balancing tree, increases memory waste in the hash table [14].

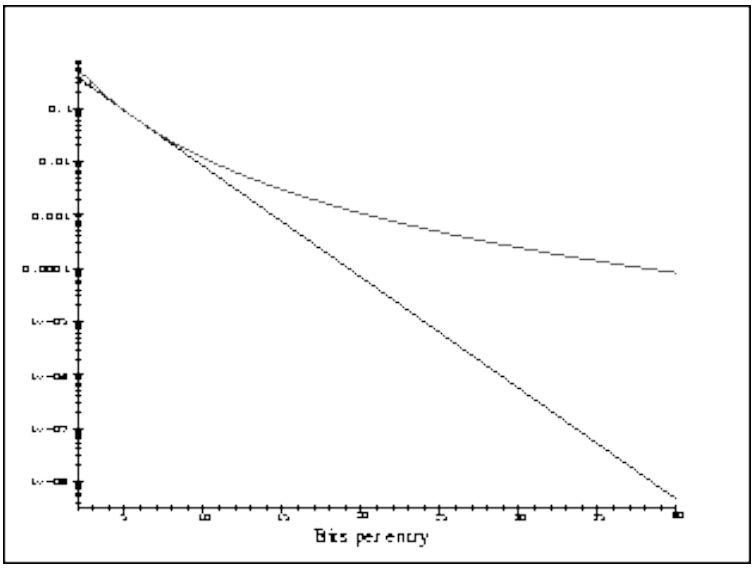

Figure 7: *The probability of false positives. The top curve is produced when four hash functions are used. The bottom curve is generated when an optimum number of hash functions are used [2].*

There are several advantages of using a bloom filter over a hash table in the caching algorithms. First, unlike the hash table, the bloom filter can store the entire universe of elements. Second, inserting an element to the bloom filter never fails because the filter never becomes full. However, the probability of false positives increases as more elements are added to the bloom filter [19]. While risking with the false positives, a bloom filter gains a space advantage over a hash table. For example, the hash table stores the contents of the block that can occupy an arbitrary amount of space depending on the block size. However, the bloom filter only stores the IDs of the blocks and their contents are not included in the filter. This data structure saves a significant amount of memory compared to the hash table. The bloom filter also has a time advantage over insert and lookup operations. Time complexity of these operations are $O(k)$ no matter how many items are in the filter. Moreover, the performance of the bloom filter can be increased significantly if it is implemented in the hardware. The reason is that the hardware allows parallelizing the bloom filter's k independent hash functions which speeds up the hash calculation operation [19].

6.4. Proposed solution

The proposed solution entails the development of a decentralized cooperative caching algorithm using a bloom filter. The architecture of the new solution would be different from that of previous solutions. In particular, each client would have local and global cache components. Moreover, the local cache stores the accurate state of the block including its content. However, the global cache stores approximate information about the block. Due to the data approximation used in the global cache, there is a chance of getting false positives and negatives. On the other hand, in the typical caching environment, the local cache is accessed more frequently than the global cache. Therefore, the rate of occasional false positives and negatives on the global cache is tolerable. Figure 8 illustrates the data structures used in the local and global cache.

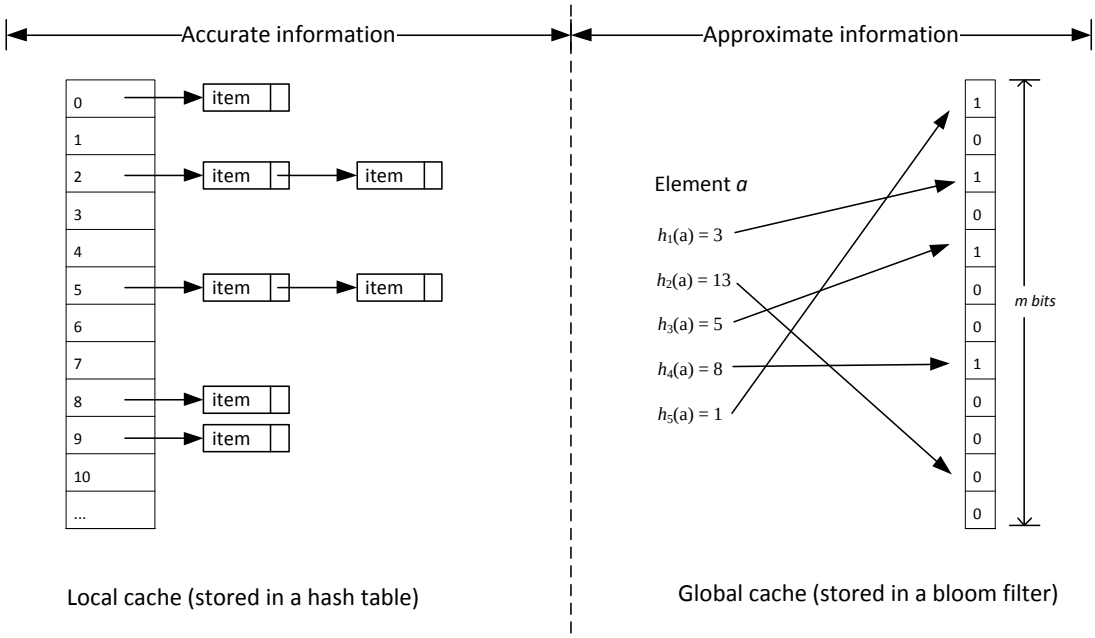

Figure 8: *The data structures used in the local and global cache.*

The proposed cooperative caching solution removes the global cache from the manager and distributes it among the clients. Thus, each client maintains its own copy of the global cache. Moreover, the client has to make sure that its copy of the global cache is up-to-date. Updating the global cache involves two-way communication among the clients. First, the client has to calculate the fresh bloom filter from its local cache and broadcast it among the other clients. The content of the global cache changes frequently, therefore the bloom filter has to be calculated and broadcasted periodically. Second, the client periodically has to accept the updated bloom filters from all of the remote clients and update the

appropriate entry in its global cache. The role of the bloom filter in the proposed solution is depicted in Figure 9.

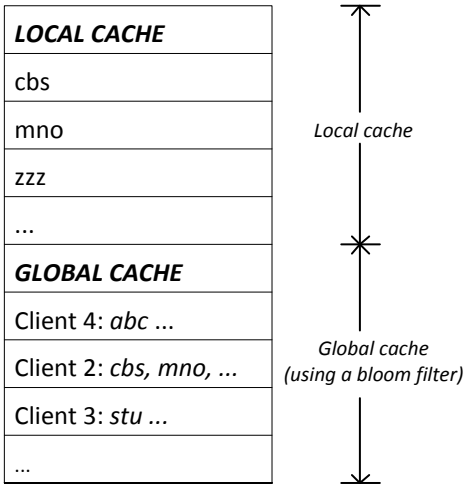

Figure 9: *The client's cache memory composed of local cache and global cache.*

The architecture of the new cooperative caching algorithm is similar to that of existing algorithms. As in the existing algorithms, the client receives a block read request from the user. If there is a local cache hit, the client serves the block from its local cache. Otherwise, the client does not contact the manager for the global cache, because the manager no longer maintains the global cache. Instead, in the new algorithm, the client looks up its copy of the global cache and makes a local decision. If there is a global cache hit, the client contacts to an appropriate remote client for the block. Otherwise, the request is redirected to the manager that fetches the block from the server's disk. Figure 10 shows the overall architecture of the proposed solution. In this figure, the client 1 receives a request for the block. First, the client 1 searches its local cache for the block, but the client cannot find the block. Then, the client looks up its copy of the global cache without contacting the manager. It turns out that the client 4 has the required block, thus the client 1 directly contacts the client 4 to read the block from its local cache.

The bloom filter plays an important role in the proposed cooperative caching algorithm. In the new algorithm, each entry in the local cache is organized as a key/value pair. The key represents the block's ID and the value denotes the block's content. However, the global cache uses a bloom filter which is only concerned with storing the keys. Thus, when the bloom filter performs a cache lookup operation, the response time is significantly faster. On the other hand, the values are stored in the local cache of remote clients. Consequently, the time to access these values becomes large because of the network latency. The remote client is contacted only if the bloom filter's lookup operation returns a positive result [1]. Figure 11 depicts where the bloom filter fits in the proposed solution.

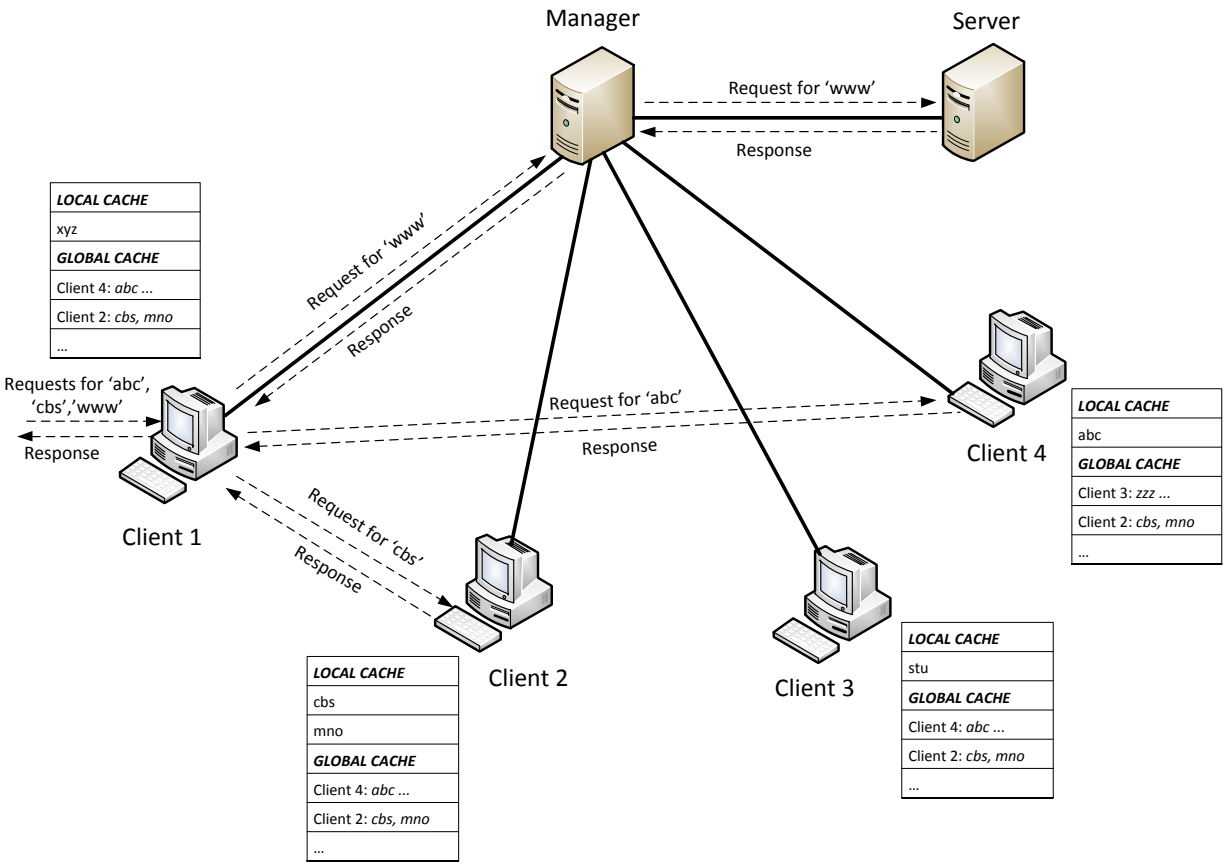

Figure 10: *The architecture of the proposed solution.*

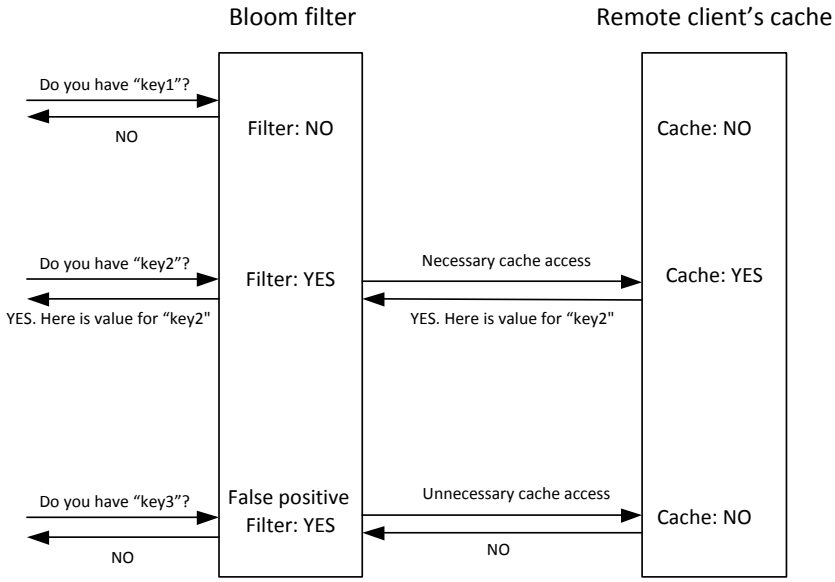

Figure 11: *The role of the bloom filter in the proposed solution [19].*

6.5. New algorithm

The results of the experiments on the new cooperative caching algorithm show that it meets the expected performance mark claimed by the hypothesis. The new algorithm performs the caching operations – cache lookup, block removal and relocation, cache consistency – using less computer resources as measured by the complexity analysis. Moreover, unlike the existing algorithms, the new algorithm uses less memory for the global cache information. Also reduced was a communication overhead among the clients which make the new algorithm scalable. The experiments are used to validate the correctness and advantages of the new algorithm.

6.5.1. Algorithm environment

The environment of the new algorithm involves input data, output data and internal data structures on which the algorithm operates. The input to the algorithm is a list of numbers of an int data type. The list of numbers represents the ID of the blocks. The algorithm returns the object of type DiskBlock, which represents the returned block, as an output. Moreover, each client maintains a local cache memory of size *M*. The cache memory can contain several cache blocks of size *m.* The maximum number of cache blocks stored in the local cache is calculated as $n = M / m$. On the other hand, the server maintains a disk that stores files made up of the blocks. The size of the block *d* and the maximum number of blocks *p* is supplied to the algorithm by external parameters [1]. In this implementation of the algorithm, $m > d$ such that $m = d + t$ where *t* is tag size.

In the new algorithm, each client maintains the local cache memory that is implemented using a HashMap data structure. The hash map's key denotes the cache block ID which is mapped into the value denoted by LocalCacheBlock object. Each entry in the local cache is a tuple (id, size, clk, isOriginal, isAlive, lifetime, diskBlock) that represents the contents of the LocalCacheBlock object. For example, id and size signifies the ID and the size of the cache block, respectively. The clk element keeps the count of the clock ticks since the block is accessed last time. Every time the local cache is accessed, the clocks of all cache blocks are decremented. If there is a local cache hit, then the clock of the appropriate cache block is reset to a default value. Next, isOriginal variable records the originality information of the block. When the block is read from the disk or it is relocated from the other client, the block is marked as an original. The copy of the original block is called a non-original block. During the block removal process, the new algorithm attempts to retain the original blocks by first removing the non-original blocks from the local cache. When the cache block's clock ticks below the allowed limit, the block becomes dead and this information is stored in the isAlive variable. In this cache implementation, when an original block becomes dead, it is

marked as a non-original. The local cache replacement policy removes the dead blocks immediately followed by the non-original and the original ones [1]. The final variable diskBlock holds the reference to the actual block that has been stored in the local cache.

Besides the local cache, the client also maintains the global cache which contains a set of bloom filter data structures. Each bloom filter is composed of the following data members: bitSet, m, n, k, noOfAddedElements, hashName and hashFunction. The bitSet represents the array that can only store either one or zero as a value in its elements. The size of the bitSet is specified by variable m which is supplied when constructing the bloom filter. Moreover, the variables n and k denote the expected number of elements to be added to the bloom filter and the number of hash functions, respectively. Unlike variable n, the variable noOfAddedElements stores the number of elements actually added to the filter at particular point in time. When an element is added to the filter, k hash functions are calculated. Each of these functions accepts the element as an input and produces values in the range $\{0 \ldots m\}$. The hashName variable represents the type of hash function, such as MD5 or SHA1, used by the bloom filter. The hashFunction represents the incremental modification used between the instances of the hashing functions. Each client periodically creates and broadcasts the summary about its local cache to remote clients. The summary is created by calculating a new bloom filter that takes the current snapshot of the client's local cache. The dead blocks are never included in the summary. The remote clients in the system receive this broadcasted summary and update their global cache by replacing the old bloom filter with a new one.

The implementation of the new algorithm also considers the hardware properties, such as the cache size and disk throughput. The assumptions about the hardware reflect the current technology as well as likely improvements in the near future. For example, the size of local cache can accept the values within range 64 – 2048 KB to keep the cache memory efficient. Also, the algorithm implements a block level granularity where the blocks make up the entire file. On the other hand, the size of the global cache is kept infinite in order to calculate the memory overhead of the algorithm. The server uses a disk as storage and the size of each disk block is assumed to be 8 KB. Moreover, the clients and the manager are connected with each other through network links. Therefore, the message latency parameter is introduced to properly reflect the RTT (round trip time) of the sent/received messages. Finally, the access time variables are used to reflect the delay incurred when reading a block from local and global caches as well as from the disk. In particular, the cache access time is composed of block seeking and fetching time. On the other hand, the disk access time is a sum of spinning time, seek time, rotational delay and fetch time [22].

6.5.2. Algorithm steps

The major steps of the new algorithm are as follows:

1. Cache lookup.
2. Block removal.
3. Block relocation.
4. Global cache maintenance.

1. Cache lookup.

When the algorithm accepts the read request for a specific block, the client first searches its local cache. If the local cache contains the requested block, the block is served immediately to the user and the algorithm terminates. Otherwise, the client looks up the global cache for the block. The global cache is made of a collection of the bloom filters that represents the latest snapshot of local cache of each remote client. The global cache lookup procedure involves searching the bloom filter for the requested block. Because the bloom filter is a probabilistic data structure, the lookup procedure may not always return an accurate answer. If the answer is accurate, the client forwards the request to the appropriate remote client that allegedly has the requested block. If fetching the block from the remote client is successful, the copy of the block is saved in the local cache and the original block is returned to the user. Otherwise, the client forwards the request to the server. Fetching a block from the server is a costly procedure and it may take a significant time compared to reading the block from local or global caches. Once the block is fetched from the server, the client saves the copy of the block in its local cache and returns the block reference to the caller. However, the local cache can only accommodate limited amount of blocks. Therefore, if the local cache becomes full, one of the existing blocks in the cache has to be removed to make room for the new block [1]. The procedure that determines the block to be removed is explained in the next step of the algorithm.

2. Block removal.

The algorithm prefers retaining the original blocks in the cache for a longer time by removing the non-original and dead blocks in the first place. Every time a block read request arrives to the algorithm, the clocks of all blocks are decremented. If the clock of a block counts down beyond the allowed threshold, the block becomes dead. If the requested block is present in the local cache, its clock is reset to a default value. Moreover, the block, which is served from the global cache or the server, also gets a refreshed clock. When the number of blocks in the cache exceeds the allowed limit, the block removal procedure is executed. The block removal process entails discarding an old block in order to make space for the new one. The block’s status is an important factor in the block removal process. For example, the dead blocks

are always removed first from the cache. If no dead block is found, the oldest non-original block is discarded next. If both of the above steps fail, an original block is removed. Prior to removing the original block, the global cache is searched for a non-original block in order to mark it as an original. If the marking attempt fails, an original block is relocated to a remote client's local cache. The purpose of such marking and block relocation procedures is to retain the original block in the cooperative caching system for longer time [1].

3. Block relocation.

The relocation procedure involves choosing a collaborator client, so that the removed original block can be saved at the collaborating client's local cache. The algorithm chooses the collaborator client that has the lowest number of original blocks. Once the collaborator is chosen, the removed original block is saved in the collaborator's cache [1]. The potential problem associated with this relocation procedure is a ripple effect. For example, the first client wants to relocate its removed block to the second client. If the second client's cache is full, it has to remove a block to make a space for the first client's relocated block. Consequently, the second client relocates its removed block back to the first client, and this procedure continues indefinitely. Thus, relocation of removed blocks back and forth creates a ripple effect which never ends. This algorithm prevents such ripple effect from happening by keeping a list of forwarding clients, so that the removed block is not relocated to the clients in the list.

4. Global cache maintenance.

The global cache maintenance procedure requires the clients to periodically exchange the summary of their local caches. In order to create a cache summary, the client has to calculate a bloom filter representation of its local cache. The bloom filter is created using the following parameters: n (the maximum number of blocks the filter can accommodate), m (the array size, also known as a load factor) and k (the number of hash functions). The values for these parameters have to be chosen carefully because they determine the correct functioning of the bloom filter. For example, the correct bloom filter ensures that the rate of false positives and negatives are kept to a minimum. The contents of the bloom filter contain the block's ID and its originality information. The dead blocks are not included in the bloom filter [1]. Once the bloom filter is calculated, it is broadcasted to all remote clients using a push mechanism. The push mechanism involves enforcing the global cache update in all remote clients.

6.5.3. Algorithm scope and limitations

Even though the new cooperative caching algorithm has an advantage over the existing ones, it exhibits some limitations. First, the new algorithm only works in block level granularity; thus, the file has to be

split into blocks in order to be stored in the server disk. Moreover, the algorithm only accepts the list of block IDs as an input, but not a whole file name. Second, the new algorithm cannot be applied to manage the caches of large CDNs that use gigabytes of cache memory. The algorithm is only tested with a cache memory of size 64 – 2048 KB, so the algorithm's performance under large scale systems is unknown. Third, the new algorithm executes sequentially and is not tailored to run in parallel. Lack of a multithreading feature prevents the clients from using the algorithm concurrently. In order to develop a parallel version of the algorithm, a special Java parallel library has to be used. On the other hand, applying parallelism into algorithm requires an external synchronization of internally used data structures. Finally, the exception handling is not fully enforced in the algorithm. Thus, it is likely to get some unhandled exceptions during the execution of the algorithm.

7. Evaluation

7.1. Software simulator

The effectiveness of the new algorithm is evaluated by comparing its performance with existing algorithms. The evaluation involves running a set of experiments to test the caching algorithms. The experiment results are used as a supporting evidence to show the correctness of the research hypothesis. In order to accurately measure the performance of the new algorithm, a trace-driven software simulator is developed and used in the experiments. The simulator accepts an input from the user and prints the output. The input data is a trace of block IDs that represents the request from the users. The output data is either a reference to the block or a null value depending on the success of request. Optionally, the simulator can also produce the outcomes of the experiments such as a log of block requests and statistics. The software simulator is developed in the Java programming language using JRE (Java Runtime Environment) version 1.6 as a runtime environment. The simulator is composed of the following components:

- ***Manager:*** an entity that performs different caching operations depending on the algorithm. For example, the manager in the N-chance algorithm forwards the block request, maintains the global cache consistency and relocates the block to another client. In the bloom filter based algorithm, the manager is merely responsible for forwarding the block request to the server. Having a single manager in the system makes it easy to measure the manager load imposed by the clients. Otherwise, a special algorithm has to be developed to balance the load among many managers

[8]. The load balancing on the managers is considered as an orthogonal issue to the research objective, thus it is not further evaluated;

- ***Client:*** an entity that is connected to the manager. The client cooperates with other clients when serving a block;
- ***Server:*** an entity with storage device that hosts the blocks;
- ***Trace file:*** a text file that contains a list of block IDs;
- ***Configuration file:*** a file that contains the simulator parameters used to configure the algorithms.

7.2. Experimental setup

7.2.1. Assumptions

The software simulator uses a number of assumptions when running the caching algorithms. Some of these assumptions are derived directly from the papers in the research area and the others are assumed based on the general knowledge of the subject. The list of assumptions is as follows:

- The block level granularity is used in all caching algorithms. Instead of receiving a file name, the simulator receives a list of block IDs that make up the entire file;
- The size of a block is fixed and no partial block allocation is allowed;
- A local cache block is composed of a tag and a block portion as shown in Figure 12;
- All block requests are read-only. The write policy of the caching algorithms is not implemented;
- Even though the simulator uses a packet switched network, the network bandwidth is not used as a parameter in the simulator. However, the network latency, which is caused by accessing the remote cache and the server disk, is controlled by an external parameter. The communication medium used among all the clients is Gigabit Ethernet and the clients are connected with each other through a Gigabit switch;

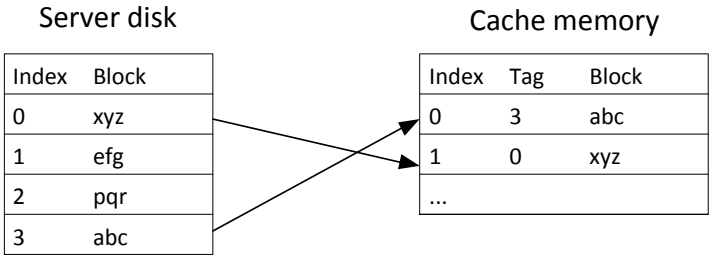

Figure 12: *The architecture of a cache block.*

- Cached blocks are stored in the primary memory rather than the disk. The bloom filter only stores the *information* about the global cache content; the *actual* blocks are stored in the remote clients caches;

- The size of a local cache is fixed, but the size of a global cache is kept infinite in order to take an accurate measure of its memory usage;
- A single Java Virtual Machine (JVM) is configured and used for the software simulator;
- No queuing delay is reflected in the block access time. Even though the caching algorithms aim to reduce the server access rate, it is assumed that queuing does not significantly influence the block access time;
- The network latency represents the RTT of sending a message from a source to a destination and otherwise. RTT excludes the amount of time the client uses to search and fetch the block from its local cache. It is also assumed that the network latency among the clients, the manager and the server is the same;
- A cache access time is the average time spent to search the required block in the cache;
- A disk access time is the average time required for the server to retrieve the required block from the disk.

7.2.2. Trace file

The software simulator uses traces of blocks as an input to the algorithm. The traces are generated randomly and they represent the block read requests from the user. The experiment uses four traces in order to test the algorithms. Figure 13 shows the statistics related to each trace.

Trace parameters	Period			
	1	2	3	4
# of Read Requests	100,000	200,000	500,000	1,000,000
# of Disk Blocks	100	100	100	100

Figure 13: *A trace period statistics. Each trace period is generated randomly using Java API 1.6.*

7.2.3. Simulator parameters

There are several parameters used by the software simulator to run the cooperative caching algorithms. These parameters are used to fine-tune the performance of the algorithms. During the simulation, the parameters are modified to reflect the specification of the underlying hardware. For example, the values of such parameters as the local cache size, the number of clients and the network latency are modified in each experiment. Figure 14 describes the environment used to evaluate the cooperative caching algorithms.

Parameter name:	Value	Unit	Description
ALGORITHM	0		Algorithm being used during the simulation: 0 – N-chance 1 – Hint-based 2 – Non-bloom filter 3 – Bloom filter
NO_OF_CLIENTS	10		Number of clients participating in the cooperative caching system
TRACE	1		Trace file being used as an input to algorithms
DETAILED_STATISTICS	0		0 – Statistics are off 1 – Statistics are on
LOCAL_CACHE_SIZE	65536	Bytes	The size of the cache in the client
GLOBAL_CACHE_BLOCK_SIZE	2048	Bytes	
TAG_SIZE	1024	Bytes	
CACHE_ACCESS_TIME	0.25	counts	The time spent to access the block in the local cache
NETWORK_LATENCY	3.85	counts	The latency experienced when sending a message from one node to the other
DISK_ACCESS_TIME	5.85	counts	The time spent to access the block in the server disk
BLOCK_SIZE	8192	Bytes	The size of the file block
RECIRCULATION_COUNT	2		The count of recirculation defines how many times the singlet block can stay in the global cache
HINT_SIZE	1024	Bytes	
HINT_COUNT	10		The number of hints each client can have in the local cache
BLOOM_FILTER_CONSTRUCTOR	1		Defines how the bloom filter is constructed: 0 – all three parameters are supplied 1 – only first two parameters are supplied
BLOOM_FILTER_LOAD_FACTOR	16	times	
NO_OF_HASH_FUNCTIONS	10		The number of hash functions used in the bloom filter
BROADCAST_INTERVAL	5	read request	The time interval that cache summaries are pushed to remote clients
LOCAL_CACHE_BLOCK_LIFETIME	15	clock counts	The number of clock counts

Figure 14: *The software simulator parameters. Each parameter is used as a knob to fine-tune the caching algorithm to its optimal performance.*

7.3. Evaluation metrics

The following metrics are used to evaluate the performance of the algorithms:

- ***Block access time:*** this metric measures the average time required to access the block. The access time is calculated based on the hit rate in the local cache, the global cache or the server disk. The algorithms that make better use of local and global caches and avoid accessing the server disk have

smaller block access time. The access times are calculated only for block read requests. Overall, when the block access time is less, the performance of the algorithm will be better [8].

- ***Manager load:*** this metric measures the client's overhead on the manager. The overhead is expressed as a number of messages generated by the clients to communicate with the manager. The messages are broken down to a block lookup, a block replacement and a global cache consistency message. Each message represents the work completed by the manager to coordinate the cooperative caching process. The less work the manager has to do on the client's behalf, the more numbers of clients it can handle [8]. Therefore, when the manager load is less, the algorithm will perform better and faster.
- ***Cache hit/miss rates:*** this metric measures the rate of reading a block from local or global caches. Cache hit rate is a sum of a local and global cache hit rates. Overall, the higher cache hit rate is a sign of better performing algorithm.
- ***Memory overhead:*** this metric measures the size of the global cache as a memory overhead. The size of the global cache is an important factor in measuring the algorithm's scalability. When the number of clients or local cache size increases, the size of the global cache also increases. Moreover, the size of the global cache directly affects the algorithm's memory overhead. Thus, the increased memory overhead may reduce the algorithm's performance.
- ***Communication overhead:*** this is calculated as a number of messages exchanged among the clients and the manager. The communication overhead is divided into block read, hint exchange and cache summary exchange messages. As the number of clients increases, the amount of messages exchanged between them also gradually grows. However, the frequent exchange of messages between the clients eventually overloads the algorithm and reduces its scalability.

7.4. Experiment results

The experiment results demonstrate the benefit of using a bloom filter in the cooperative caching algorithms. The performances of new and existing algorithms are compared based on several metrics. The configuration parameters of the software simulator are adjusted regularly based on the requirement of the experiment. Most experiments involved testing the sensitivity of the algorithm's performance for future possible hardware upgrades. The most frequently modified configuration parameters are the local cache size, the number of clients and the network latency. The explanations accompanied to the experiment results use shorter names for the caching algorithms. For example, hint-based algorithm is referred to as hint algorithm. On the other hand, non-bloom filter based algorithm and bloom filter based algorithms are simply called non-bloom and bloom algorithms, respectively.

7.4.1. Block access time

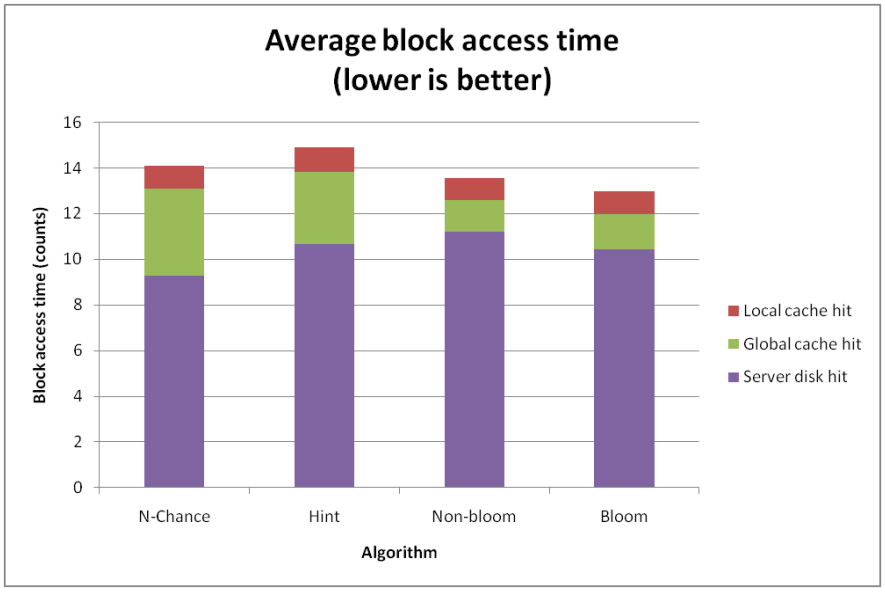

Figure 15: *The average block access time. This figure shows the average block access time for four cooperative caching algorithms. The segments of the bars show the fraction of block access time contributed by the hits of local cache, global cache and server disk.*

Figure 15 shows that the average block access time for the bloom algorithm is slightly lower compared to the rest of the algorithms. One of the reasons for such result is that the bloom algorithm manages the contents of the local cache more efficiently due to an improved local cache replacement policy. This policy attempts to retain the original blocks in the local cache for a longer time. Thus, in the bloom algorithm, the client accesses the global cache and the server disk less frequently. Consequently, the access time for the block is reduced. However, the rest of the algorithms use primitive LRU policy which does not give a preference to the master or singlet blocks. While all algorithms spend similar amounts of time to access the local cache, their global and server disk hit rates are different. For example, the N-chance and hint-based algorithms' block access time have larger global cache hit portion. Because once the singlet or master blocks are removed from the local cache, these blocks are relocated to the other clients. Moreover, the clients constantly update the global cache with changes in their local cache. Such a frequent update results in a more accurate global cache which increases the global cache hit rate. However, the bloom algorithms' global cache hit fraction is less than the other two algorithms. This is because the global cache contents are updated less frequently in the bloom algorithm. Therefore, the slow frequency of broadcasting the cache summary may cause the global cache to become obsolete very quickly. This reduces the global cache hit rate in bloom and non-bloom algorithms. However, decreasing the broadcast time interval could increase the global cache hit rate. Moreover, this optimization could reduce bloom algorithm's block access time even further.

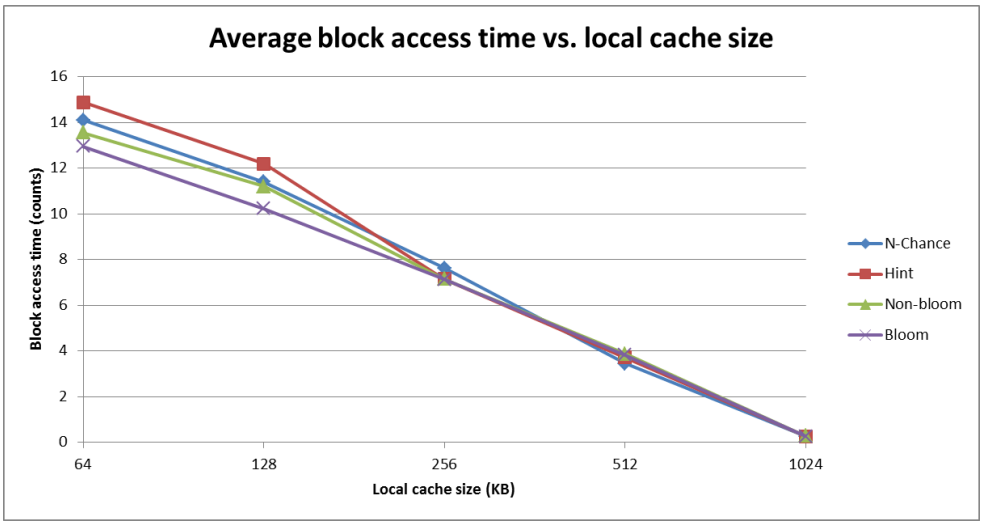

Figure 16: *The sensitivity of the block access time to the variation in the local cache size.*

Figure 16 demonstrates the result of the experiment on the sensitivity of the block access time to the variation in the local cache size. This variation is one way of measuring the scalability of the algorithms. The experiment measures the average block access time as the local cache size varies from 64 KB to 1024 KB. The rest of the configuration parameters in the software simulator are kept fixed. According to Figure 16, as the local cache size is increased, the block access time is gradually reduced for all algorithms. Thus, when the local cache size is larger, less time is spent to access the blocks. There are two reasons for such phenomenon. First, the large cache size decreases the local cache miss rate because the cache is large enough to accommodate more blocks. Consequently, the local cache hit rate is increased because most of the block requests are satisfied by the local cache due to a locality of reference concept. Thus, the client contacts the global cache and server disk less frequently. Also, the rate of accessing the global cache and the server disk is reduced. Second, the large local cache increases the size of the global cache, thus the access rate for the server disk reduces. However, once the local cache size is set to 256 KB, the block access time for all algorithms becomes nearly identical. This is because the working set size of the client applications becomes equal to the aggregate size of the client local caches [8]. The same trend continues when the local cache size is increased to 512 KB and again to1024 KB.

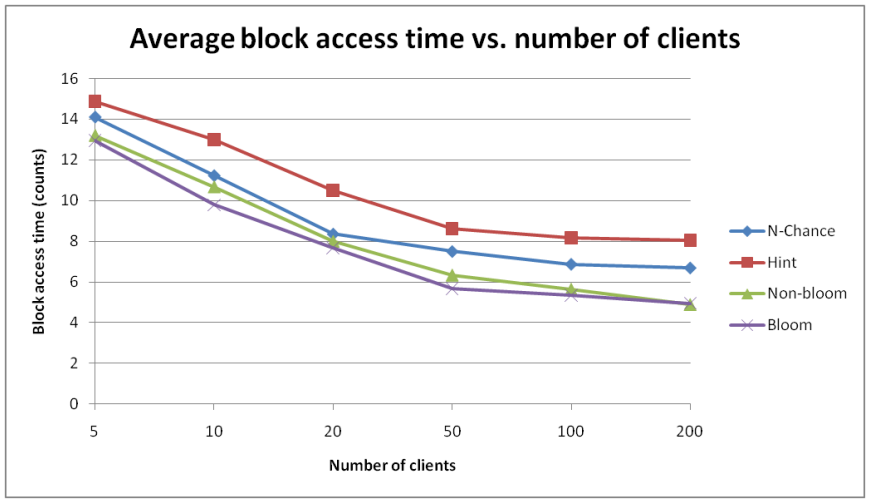

Figure 17: *The variation in the block access time as the number of clients increase.*

The scalability of the algorithms can also be tested by increasing the number of clients in the cooperative caching system. Figure 17 shows the result of the experiment where the number of clients changed from 5 to 200. According to the figure, the block access time gradually decreases as more clients are connected to the system. When the number of clients increases, the global cache size also increases because of the aggregate size of the local caches. Thus, the client contacts the server less frequently because the global cache becomes large enough so that more blocks can be accommodated. Consequently, the global cache hit rate increases and this reduces the block access time in all algorithms. However, once the number of clients reaches fifty, the block access time reduces more slowly. One reason for such behavior could be that the working set size of the client applications starts to approach the size of global cache. Another reason could be due to the network latency experienced when reading the block. However, the value of network latency is kept constant during the experiment. Thus, block access time approaches the network latency cutoff value but never gets below it.

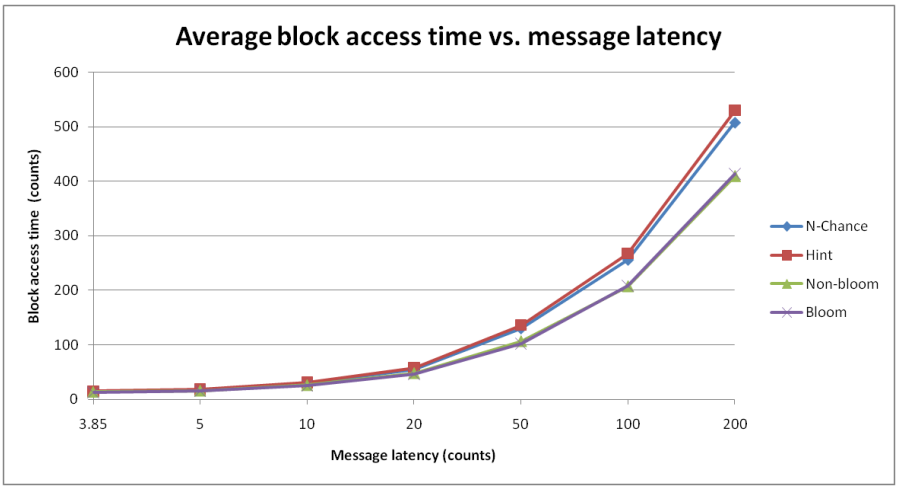

Figure 18: *The variation in the block access time as network latency increases.*

Figure 18 outlines the result of the experiment on the sensitivity of block access time as the network latency increases. The value of the network latency runs from 3.85 to 200 counts. This value represents the latency occurred in local area networks (LAN) and wide area networks (WAN). When the network latency is low, the block access time in all algorithms is similar. One reason for such result could be a small network latency value. For example, the small network latency limits the advantages of the bloom algorithm regardless of its decentralized architecture [8]. Moreover, the small network latency means that the clients and the manager are located very close to each other, so there is little delay in accessing the manager and the server. However, as the network latency increases, the block access time in the bloom algorithm grows slower than other algorithms. This is due to the fact that the client and the manager are moving further apart from each other. However, the centralized algorithms have to use the manager more often, so the block access time in these algorithm grows faster compared to the bloom algorithm. Therefore, the bloom algorithm has a potential to perform better than the existing algorithms over the networks ranging from LAN to WAN [8].

7.4.2. Manager load

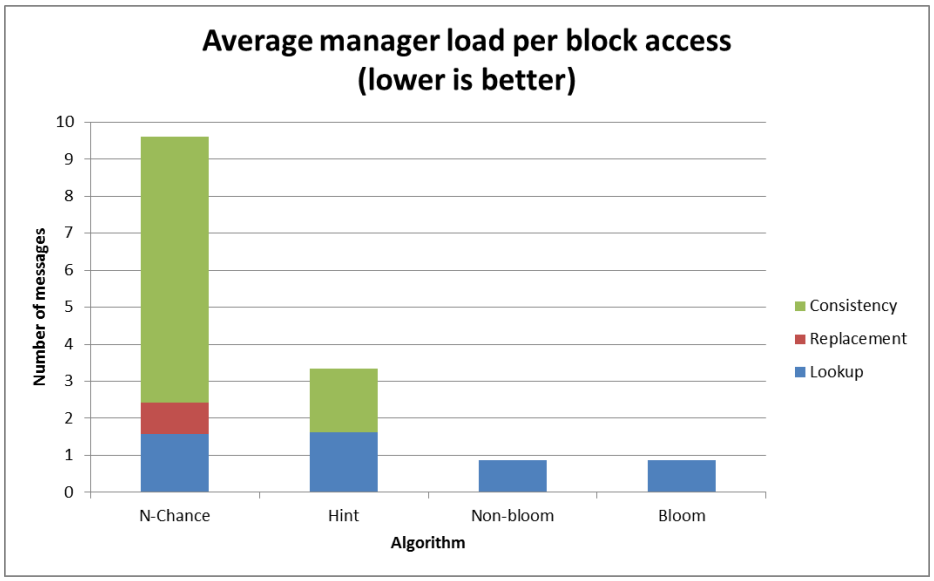

Figure 19: *The average load imposed on the manager by each algorithm. The load is defined as number of messages sent and received by the manager. The manager load is broken down to consistency, replacement and lookup messages.*

Figure 19 shows the measurement of the load imposed on the manager by each algorithm. The manager load is broken down to consistency, replacement and lookup messages. According to the figure, the N-chance algorithm has the highest value for the manager load compared to the rest of the algorithms. This is because the centralized nature of N-chance algorithm requires contacting the manager frequently to maintain the global cache consistency. Another reason for such high value is because a client constantly updates the manager with the change in its local cache content. It is also worth mentioning that in the N-chance algorithm, the number of consistency messages dominate the manager load these messages are used for the maintenance of the global cache. Moreover, the replacement messages also contribute to the manager load because the block removed from the local cache is forwarded to another client through the manager. In the case of the hint-based algorithm, the manager load is reduced by a factor of three. Regardless using the local hints, the algorithm still requires a frequent contact with the manager in order to maintain the facts in the global cache. However, the bloom algorithm decentralizes the manager's responsibility which makes the algorithm more scalable. Thus, the manager load is significantly low in the case of the bloom algorithm. Another reason for such a low value is because the global cache is locally stored in each client, so clients do not contact the manager at all. In conclusion, the measurement of the manager load gives an insight into the scalability of each algorithm.

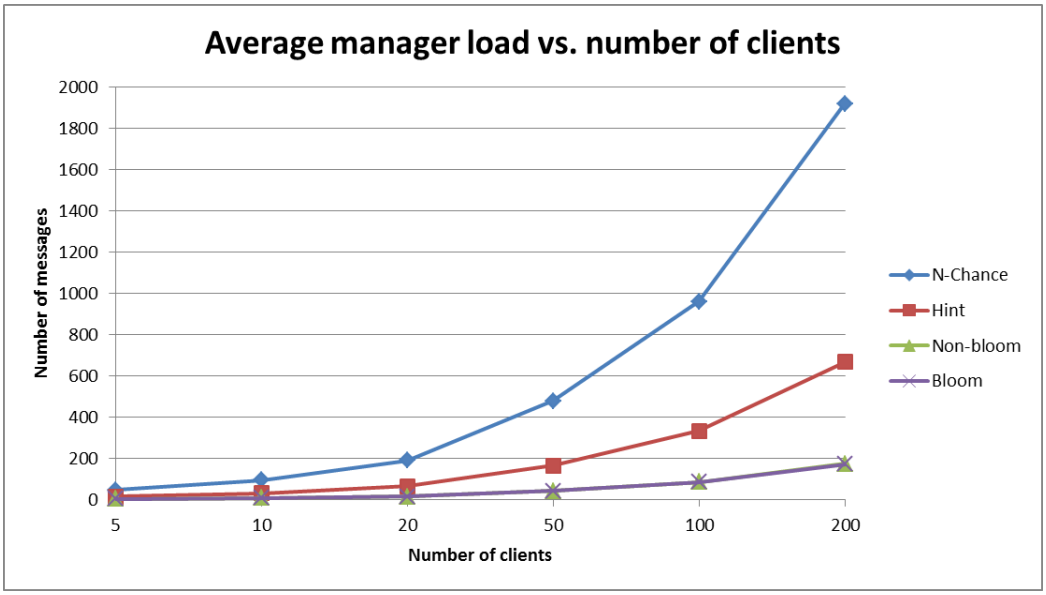

Figure 20: *The variation in the manager load as the number of clients increases.*

While the previous result of experiment on the manager load looks promising, it is also important to understand how this result is maintained as the number of clients increase. Figure 20 shows the result of the experiment on scalability of the algorithms where all simulator parameters is kept fixed except the number of clients. According to the figure, the manager load on bloom and non-bloom algorithms grows slower than the rest of the algorithms. The reason for such behavior is due to the decentralized architecture of these algorithms. The decentralized architecture keeps the communication among the clients and the manager low regardless of the number of clients connected to the system. For example, even though the number of clients reaches to 200, the clients under the bloom algorithm contact the manager only if they experience global cache. However, the N-chance and hint-based algorithms show a gradual increase of the manager load as more clients are connected to the system. The reason is that algorithms require the clients to contact the manager to maintain the global cache consistency. Moreover, as the number of clients increase, the amount of messages exchanged among the clients and the manager becomes enormous. Thus, the performance of the manager may eventually degrade which reduces the algorithm's scalability.

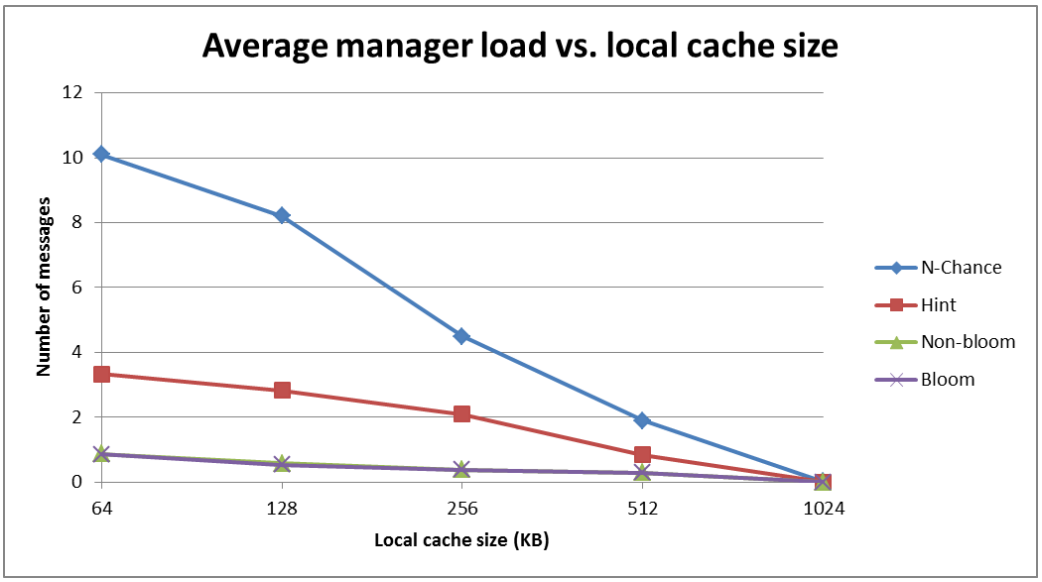

Figure 21: *The variation in the manager load as the size of the local cache changes.*

The scalability of the algorithms can also be tested by adjusting the size of the local cache. Figure 21 shows the result of such testing where the size of the local cache varied from 64 KB to 1024 KB. According to the figure, the manager load is reduced as the local cache size is increased. This behavior is due to the cache size and locality of reference principle. As stated previously, the large size of a local cache allows storing more blocks. Consequently, the clients would experience more local cache hits and they contact the manager or the server less frequently. Therefore, fewer messages are sent to the manager to lookup the global cache or to relocate the removed block. Once the local cache size reaches 1024 KB, almost all block requests are served from the local cache. In general, the performances of the non-bloom and bloom algorithms are better than others when the local cache size is calibrated between 64 – 512 KB. The reason for such behavior is because the clients make local decisions on caching operations and rarely contact the manager. Therefore, having less local cache size does not necessarily increase the number of messages sent to the manager.

7.4.3. Cache hit/miss rates

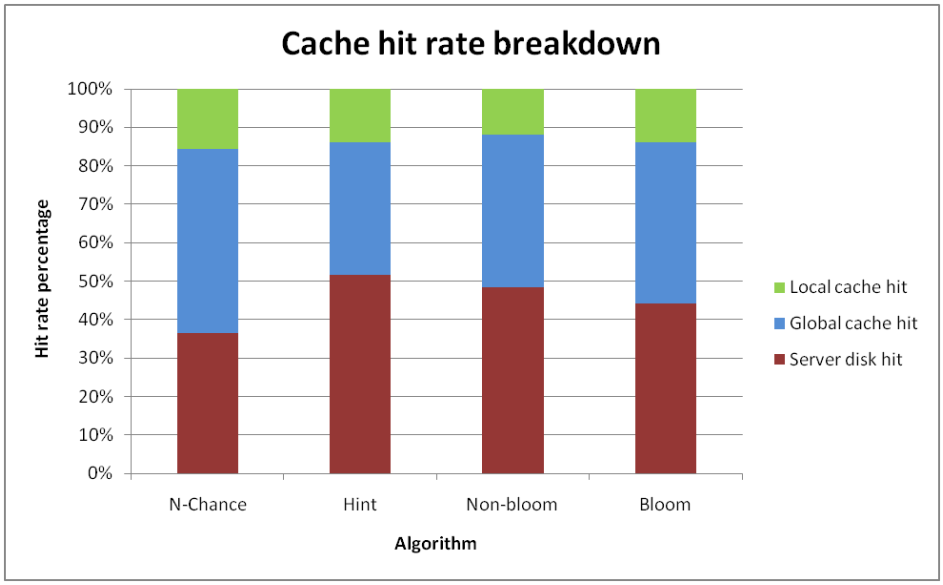

Figure 22: *The breakdown of the cache hit rate in each algorithm. The cache hit is composed of the local cache hit, global cache hit and server disk hit rates.*

Figure 22 presents the result of an experiment on hit percentages at the different layers of the memory hierarchy. This result provides an additional insight into the performance of the cooperative caching algorithms. In the figure, the bar represents the total cache hit rate of each algorithm. Total cache hit rate is composed of local cache hit, global cache hit and server disk hit rates. According to the figure, all algorithms have similar local cache hit rate. Moreover, the global cache hit rates are also similar all caching algorithms. This is because these algorithms constantly keep the global cache updated which increases its hit rate. For example, in the N-chance algorithm, the clients constantly contact the manager to update the global cache state. On the other hand, the bloom algorithm keeps the global cache consistent by frequently broadcasting the local cache summaries to all clients in the system. Therefore, the up-to-date global cache is likely to be accessed more often than the server disk. However, the hint-based algorithm has the smallest global cache hit rate among all algorithms. This is because the algorithm maintains the global cache using hints in the clients that may not be accurate at all times. Thus, the inaccurate hints slightly decrease the global cache hit rate.

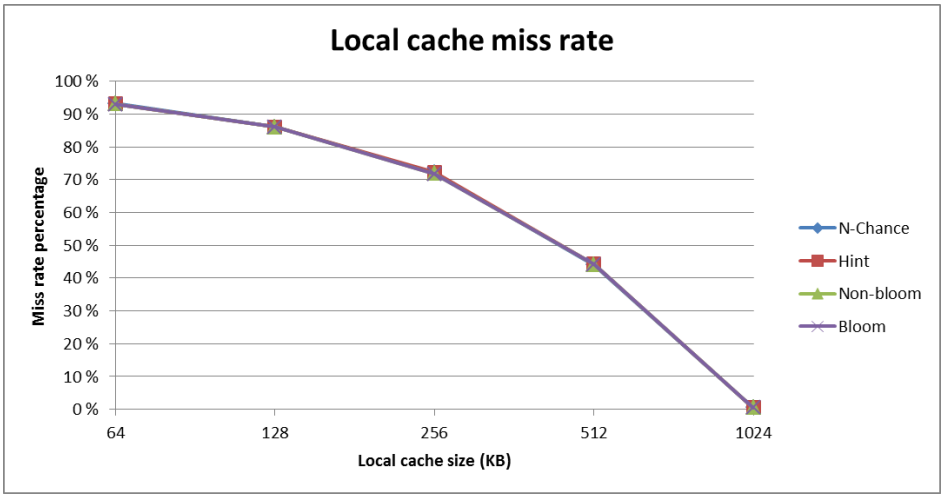

Figure 23: *The variation of the local cache miss rate as the size of local cache increases.*

The next experiment demonstrates the correct operation of all caching algorithms (Figure 23). The correctness can be verified by increasing the local cache size. The expected phenomenon should show that as the local cache size increases, the local cache miss rate gradually reduces. According to the Figure 23, all algorithms maintain the same local cache miss rate as the local cache size is increased. One reason for such behavior is due to the locality of reference concept. According to this concept, the same or nearby blocks in the memory will be referred to in a short period of time. Thus, keeping more blocks in the cache decreases the cache miss rate. Another reason could be that as the cache size increases, the local cache would have enough space to accommodate all required blocks. Therefore, it becomes unnecessary for the client to access the global cache or the server disk for the block. Moreover, when the local cache size reaches 1024 KB mark, almost all of the blocks become available in the local cache. This defeats the purpose of having a global cache and server disk. However, in practice, the size of the cache should be kept small in order to retain its efficiency.

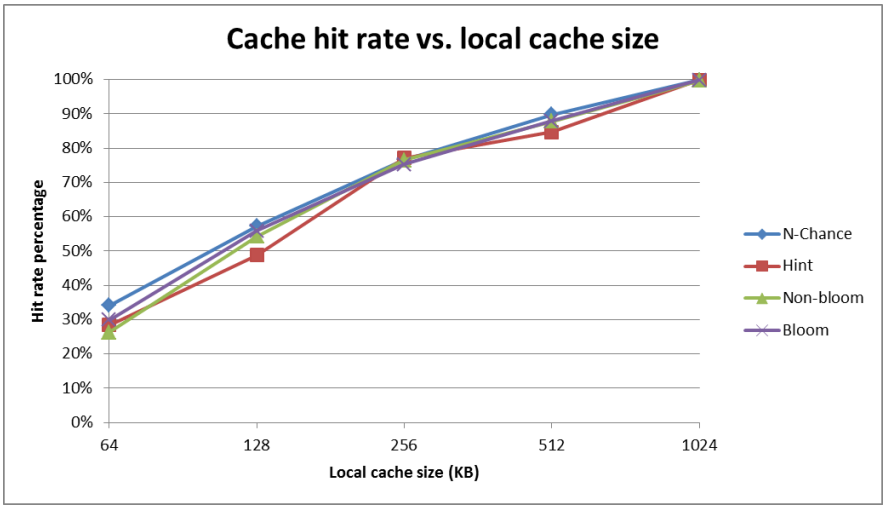

Figure 24: *The variation in the cache hit rate as the local cache size increases. In this experiment, the cache hit rate is composed of local and global cache hit rates.*

The cache hit rate of the algorithms is also affected by increasing the local cache size. Figure 24 shows the outcome of the experiment on the cache hit rates. In this experiment, the size of a local cache is varied from 64 KB to 1024 KB. According to the results, as the local cache size increases, the cache hit rate also gradually climbs. One reason for such behavior is the change in cache size. Increasing the size gives enough space for the cache to accommodate more blocks. Consequently, an algorithm never uses its local cache replacement policy to remove the older blocks. This causes most of the user requests to be served directly from the local and global caches rather than the server disk. Another reason for an increase in the cache hit rate could be the size of the client working set. In particular, when the working set size approaches the aggregate size of local and global cache, the cache hit rate increases. Thus, when the local cache size reaches 1024 KB, the cache hit rate becomes almost 100%. The reason is because the cache contains all the blocks in the system. Moreover, no blocks are removed from the cache and they remain in the cache forever. Thus, any read request always returns the block from either the local or the global cache.

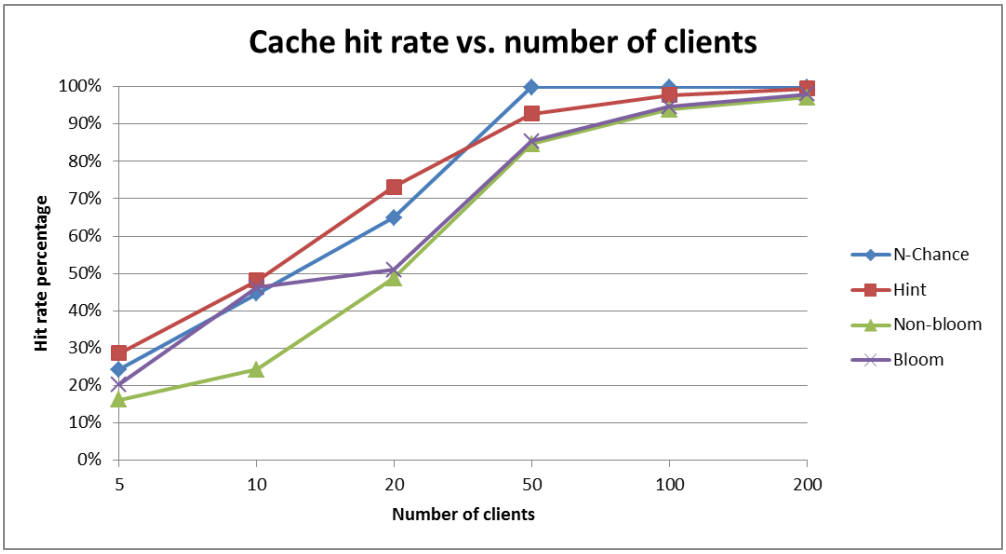

Figure 25: *The variation in the cache hit rate as the number of clients increases. In this experiment, the cache hit is calculated as a combination of local and global cache hit rates.*

Figure 25 demonstrates the result of the experiment on the scalability of the caching algorithms. This experiment entails the measurement of the cache hit rate as the number of clients change from 5 to 200. According to Figure 25, the cache hit rate, which is a combination of local and global cache hit rates, climbs gradually as the number of clients change from 5 to 50. For example, when the number of clients is 50, the cache hit rate in the N-chance algorithm reaches 100% mark. But in the case of other algorithms, the cache hit rate remains between 80% - 90%. The reason for such reaction of the algorithms is simple. Increasing the number of clients results in a larger global cache size which increases the cache hit rate. For example, in the case of the N-chance algorithm, the global cache contains all the blocks in the system when the number of clients is 50. This is because the global cache is accurate in the N-chance algorithm, so 50 clients are enough to have a global cache that contains all blocks. However, the cache hit rate in other algorithms does not reach 100% until the number of clients is 200. The reason for this is because these algorithms do not maintain an accurate state of the global cache. For example, hints in the hint-based algorithm are not accurate at all times. Moreover, the global cache in the bloom algorithm is only updated periodically. Thus, an inaccurate global cache state usually exhibits lower hit rate.

7.4.4. Memory overhead

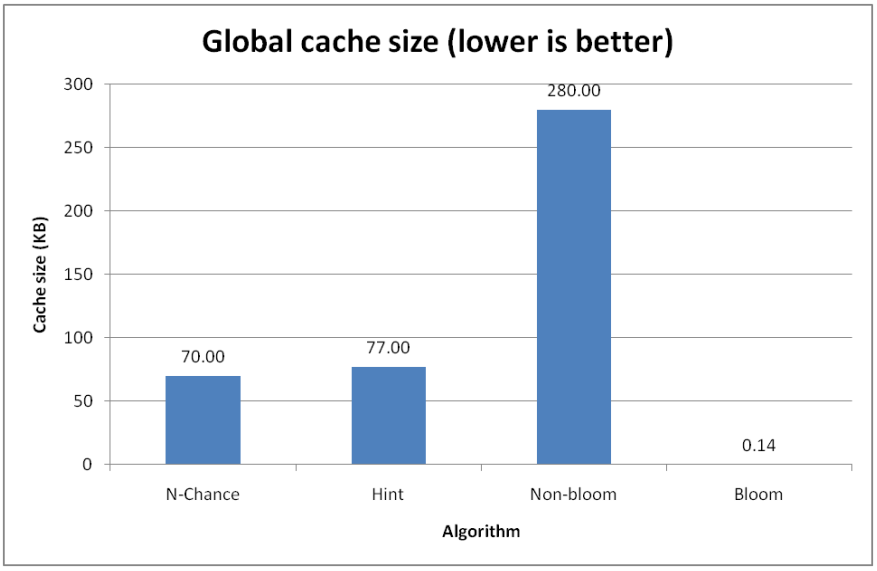

Figure 26: *The memory overhead in the cooperative caching algorithms.*

Figure 26 shows the result of the experiment on the memory overhead of the algorithms. In this research, the global cache size is used as a measure to calculate the memory overhead. The reason is that the global cache size is variable and its capacity depends on the clients' local cache size. According to Figure 26, the bloom algorithm's memory overhead is significantly low compared to that of other algorithms. This is because the new algorithm uses a bloom filter for the global cache. Moreover, the size of the filter is fixed and it does not change as more block information is added to the global cache. On the other hand, the memory overhead is the highest in case of non-bloom algorithm. The reason for such a huge difference in the memory overhead is due to the method used to store the blocks in the global cache. In this research, all caching algorithms, except the non-bloom algorithm, only use the information about the blocks to build the global cache content. The actual block content is kept at the client's local cache. However, the non-bloom algorithm uses a global cache to store the exact copy of a block. Due to these design decisions, the memory overhead is high in the non-bloom algorithm, whereas this value is significantly low in the rest of the algorithms.

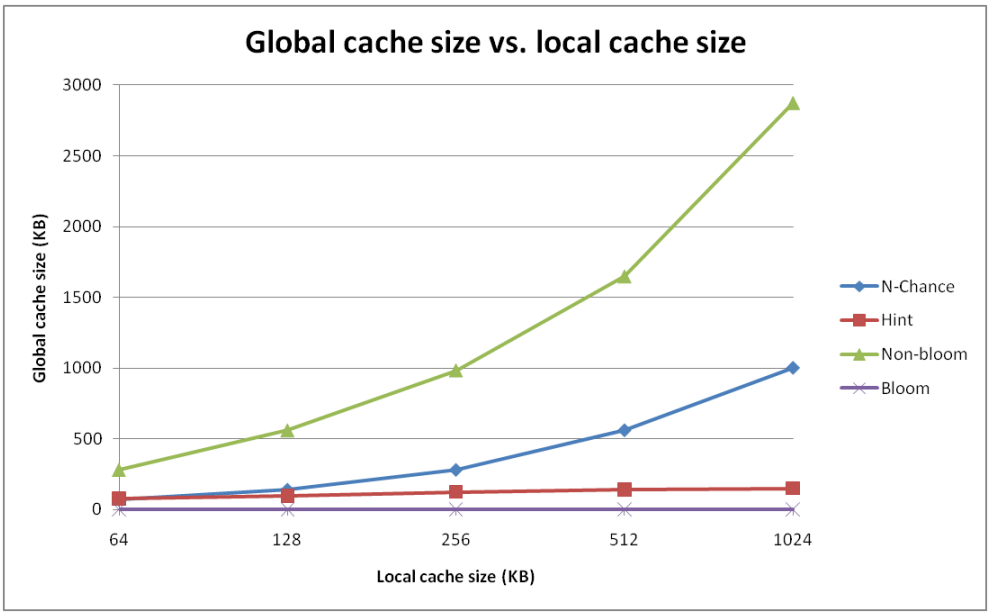

Figure 27: *The sensitivity of the memory overhead to the variation in the local cache size.*

The next experiment also tests the algorithm for scalability where the local cache size is increased from 64 KB to 1024 KB. The experiment result is shown in Figure 27. In general, the size of the global cache directly depends on the size of the local cache. Therefore, when the local cache size is increased, so does the global cache size. For example, in case of the non-bloom algorithm, the global cache size increases exponentially when the local cache size reaches the 1024 KB mark. This is due to the fact that the algorithm stores the block contents in the global cache. Thus, the system with more clients requires a large global cache. Next, in the case of N-chance algorithm, the global cache size grows slower than that of the non-bloom algorithm. However, the algorithm still requires more memory space for the global cache as the number of clients increases. The reason is that the manager tries to maintain the most recent version of the global cache which is made up of the information about all blocks in the system. Thus, the global cache in the N-chance algorithm still consumes more memory. On the other hand, the hint-based algorithm's global cache size remains stable as the local cache size increases. This is due to the fact that the global cache keeps the information only about the master blocks. Thus, the amount of master blocks in the global cache consumes less memory space. Finally, the bloom algorithm demonstrates immunity to the increase in the local cache size. The reason is that the algorithm uses a fixed size data structure for the global cache state. Therefore, the size of the data structure does not change as more blocks are added to the global cache.

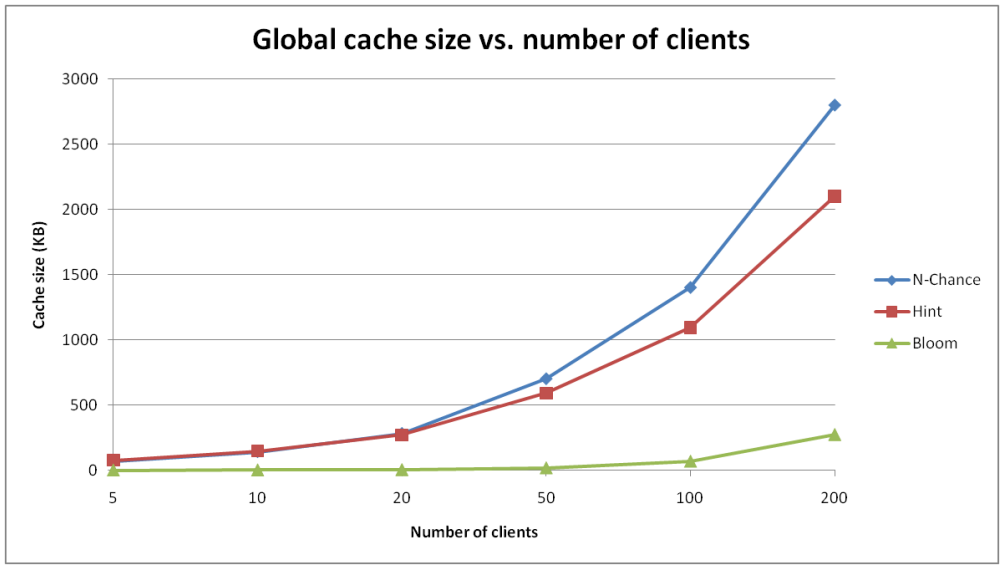

Figure 28: *The sensitivity of the memory overhead to the change in the number of clients.*

Figure 28 shows the result of another scalability test on caching algorithms. In this experiment, the number of clients was an important factor to determine the algorithm's scalability. According to the figure, the number of clients in the system directly affects the memory overhead. For example, when the number of clients increases from 5 to 200, the global cache size climbs to 3000 KB and 2200 KB mark for the N-chance and the hint-based algorithms, respectively. Such exponential growth of the memory overhead indicates that these algorithms are not scalable and their performance is reduced over time. One reason for such behavior is because these algorithms experience increased block access time and manager load when the number of clients increases. Consequently, the algorithms would exhibit low performance. Another reason is that both the N-chance and hint-based algorithms use a variable size global cache which can grow infinitely. Hence, when the number of clients increases, the amount of blocks occupies more space in the global cache. On the other hand, the experiment results show that the bloom algorithm's global cache size remains relatively stable until the number of clients reaches 50. The reason for such behavior is because of the fixed size of the bloom filter. However, the global cache has one bloom filter for each client. Thus, when the number of clients increases over 50, the global cache size becomes noticeable in the graph.

7.4.5. Communication overhead

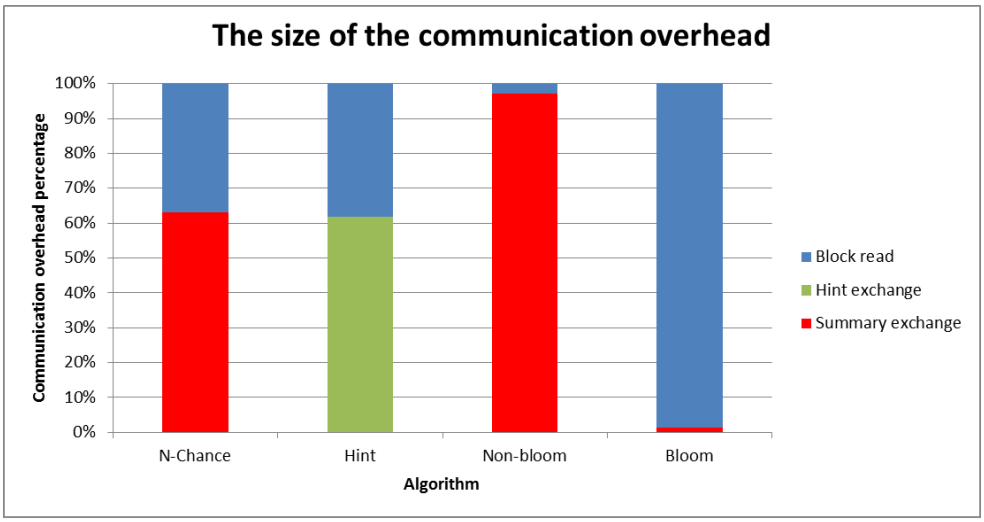

Figure 29: *The size of the communication overhead in the algorithms. The communication overhead is calculated by multiplying the message size by the number of messages exchanged between the clients and the manager.*

Figure 29 shows the result of the experiment conducted to measure the size of the communication overhead in the algorithms. According to the figure, both the N-chance and hint-based algorithms use about 40% of their communication messages to read the blocks from the memory. The rest of the messages are used to update the global cache through summary messages or hint exchanges. Moreover, the experiment results show that the N-chance algorithm is more concerned with exchanging the messages with the manager to keep the global cache consistent. This is because the consistency of the global cache is a priority in this algorithm. This priority is also valued in the case of hint-based algorithm. Thus, the hint and fact exchanges happen more frequently compared to reading a block. Next, the non-bloom algorithm exchanges the block contents among the clients; therefore, the summary message dominates the algorithm's communication overhead. In case of the bloom algorithm, 98% of the communication overhead is used to read blocks from the memory, but only 2% is used to update the global cache. The reason is because the algorithm uses a fixed size bloom filter for global cache update. Moreover, the global cache updates happen periodically. Thus, the summary exchange messages produce significantly less communication overhead.

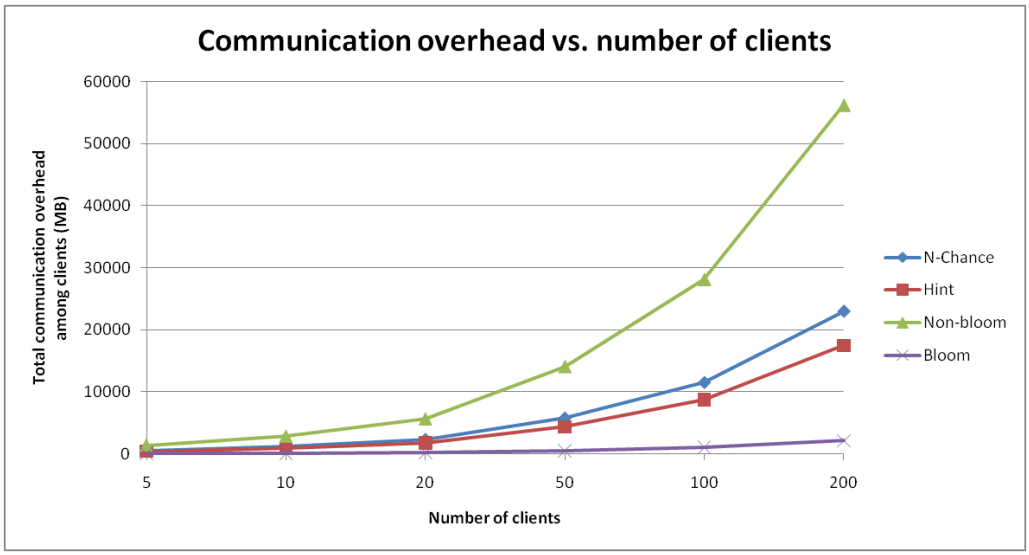

Figure 30: *The variation of the communication overhead as the number of clients increases.*

Figure 30 shows the results of the experiment that calculates the size of the communication overhead between the clients and the manager. According to the experiment results, small number of clients do not significantly influence to the communication overhead of the algorithms. When more clients are added to the system, the communication overhead in all algorithms gradually starts to increase. For example, in the case of bloom algorithm, the amount of the cache summaries exchanged among the clients remains relatively small due to the fixed size of the bloom filter. However, the non-bloom algorithm quickly increases its communication overhead as the number of clients increase. In the cases of hint-based and N-chance algorithms, an increase in the rate of communication overhead is not as high as the non-bloom algorithm. The reason for such different behaviors is due to the architecture of the algorithms. For example, in the N-chance, bloom and hint-based algorithms the clients only exchange the information about the blocks, but not the block contents. Moreover, the block information can be in the form of a hint or a bloom filter and its size is relatively smaller than the block content. However, in the non-bloom algorithm, the clients exchange the block contents. Therefore, the increase in the number of clients prevents the non-bloom algorithm from being scalable.

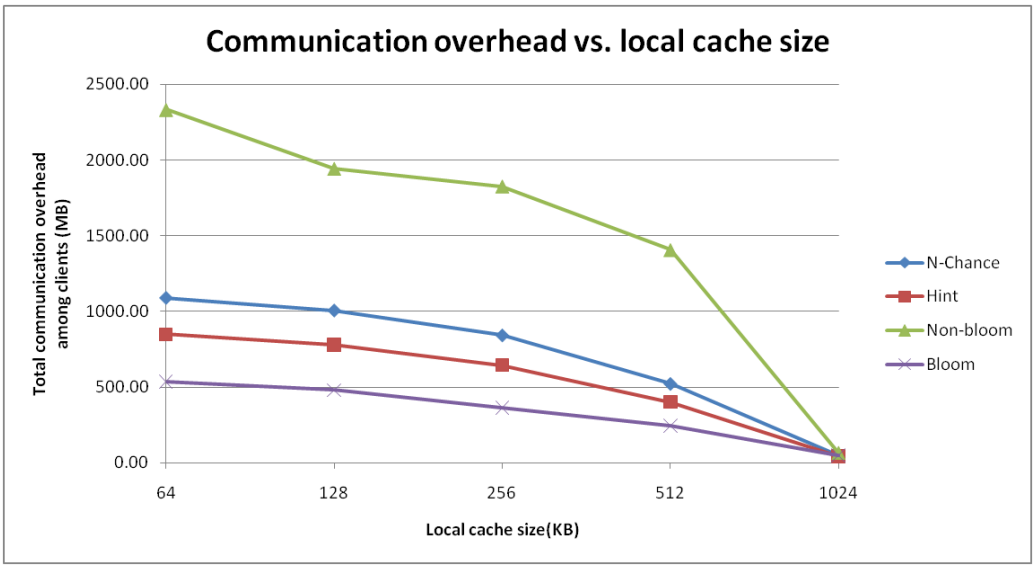

Figure 31: *The variation of the communication overhead as the local cache size increases.*

The final round of the experiment results are presented in Figure 31. This experiment measures the size of the communication overhead as the local cache size is changed from 64 KB to 1024 KB. According to the figure, all caching algorithms demonstrate similar communication overhead pattern. The reason for such pattern is simple: as the local cache size increases, the client starts to accommodate more blocks in its local cache. This means that the majority of the block requests are served from the local cache. Consequently, the local cache replacement policy removes fewer blocks from the cache and the block relocation procedure takes place less frequently. Thus, the size of the communication overhead is reduced due to less frequent message exchanges among the clients. On the other hand, an individual result of communication overhead is different in each algorithm. For example, the non-bloom algorithm has a relatively high communication overhead because it deals with exchanging and reading the block contents. The rest of the algorithms merely exchange the block information in the form of a bloom filter or a hint which is relatively small in size. Thus, these algorithms demonstrate lower communication overhead.

7.5. User manual

This section outlines the steps necessary to setup the software simulator in order to reproduce the experiment results. The system requirements for the simulator include the Java JDK (Java Development Kit) version 1.6 or later and Eclipse IDE (Integrated Development Environment). The source code of the simulator is developed using Eclipse IDE and the code files are grouped into a project. Therefore, using the Eclipse IDE makes it easy to understand the project structure and run the simulator. On the other

hand, the Java JDK is required to compile the source code files into binary files. The operating system platform is not important to run the Java program as long as there is a JVM for it. Once the project is opened using Eclipse IDE, the algorithms have to be supplied with an input data in the form of trace files. The trace files are generated using random classes in Java API (Application Programming Interface). The sample trace files used in the experiments are included as a part of the project. One of the important components of the simulator is its configuration file. The configuration file is a simple text file that contains a list of parameters. A user of the simulator can set all parameters in the configuration file in order to adjust the performance of the algorithms. The range of values each parameter can accept is included in the thesis report.

8. Deliverables

The main deliverables of this thesis are:

- ***Thesis proposal*:** outlines the overview of the current thesis;
- ***Thesis report:*** a comprehensive report that contains the motivation behind the thesis, the problem definition, the research hypothesis, the detailed cooperative caching architecture, the related works published in the literature and the experiment results. The report also includes the user manual to setup and run the software simulator;
- ***Simulator software*:** written in Java programming language (including the source code);

9. Thesis limitations

The results of the experiments and the theoretical analysis confirm the validity of the research hypothesis. While the current research achieved its objectives, there are some caveats related to the proposed solution that may disprove the hypothesis. For example, the bloom filter only stores the approximate state of the global cache. This may result in false positives and negatives such that the global cache lookup operation

may not return an accurate answer. Inaccurate answers contribute to the local and global cache misses. Consequently, this increases the block access time and reduces the performance of the new algorithm. Therefore, the bloom filter has to be created accurately using the proper values for the filter's parameters. Moreover, the solution proposed in this research does not implement the writing policy. If the writing policy is integrated into the new algorithm, the contents of the cache memory and the disk would have been more consistent. However, it is unlikely that writing policy changes the outcomes of the experiments. The reason is that the writing policy does not interfere with the block reading policy; therefore, it should not affect the cache hit/miss rates. Moreover, the writing policy is orthogonal to the research objectives; thus, it is unlikely that implementation of the policy would have disproved the research hypothesis. Another shortcoming of this research is related to the input data used in the experiments. The input data is synthetic meaning that the data is randomly generated which does not reflect the real world block access pattern. However, using real world traces would increase the credibility of the experiment results. Moreover, the real world traces are likely to affect to some extent the results of cache hit/miss rates, the block access time and the manager load.

10. Future work

While the current research achieved its objectives, there are some other avenues that have to be explored in the future. One of the future works entails implementing a cache writing policy in the new algorithm. According to this policy, when the content of the cache is changed, the updated blocks are written back to the server's disk. Thus, the consistency of the cache memory and the disk is maintained. Another extension to the current research is to implement an offline caching algorithm. The offline algorithm can be used in the experiments to measure the upper performance bound of the online algorithms. Such measurement would help to optimize the performance of the existing algorithms. Moreover, the list of future works can be extended with implementation of the file based granularity. The file based granularity would allow the caching algorithms to accept the entire file name as an input data. Finally, implementing a multithreading would improve the performance of the new caching algorithm. For example, the computation of the bloom filter can be parallelized so that each hash functions are calculated by separate threads. Overall, the future work for the current research involves the implementing many features that would extend the capability of the new algorithm.

11. Conclusion

A cooperative caching is a technique that enables the clients to access each other's local cache when the client experiences a cache miss. This flexibility improves the efficiency of the local and global cache replacement policies. The cooperative caching system usually consists of the manager, the clients and the server. Moreover, the system needs an algorithm to coordinate the contents of the local and global caches. The level of coordination is an important factor and this is the main difference between various cooperative caching algorithms. For example, the N-chance algorithm implements a higher level of coordination by using the central manager that maintains the global cache. However, the algorithm suffers from a scalability problem due to its centralized approach. On the other hand, the hint-based algorithm distributes the portions of the global cache to the clients as hints. However, due to inaccuracy of the hints, the central manager has to maintain the facts in the global cache. Even though such arrangement reduces the load on the manager, the hint-based algorithm is still remains vulnerable to the communication and memory overhead problems.

The current research presents the bloom filter based algorithm that addresses the problems of existing solutions. The new algorithm removes the manager's responsibility of maintaining the global cache. Instead, the entire global cache is distributed to the clients so that the each client maintains its own version of the global cache. The content of the global cache is composed of a set of bloom filters. This allows the clients to query the global cache and make decisions locally. The bloom filter also simplifies the implementation of the global cache replacement policy which does not involve the manager. The experiments on the new algorithm show that the block access time is reduced as a result of distributing the global cache to clients. Moreover, the load on the manager is also decreased due to a decentralized approach of the algorithm. The reason is because the manager dropped the responsibility of maintaining the global cache; thus, it can handle more clients in the system. Consequently, the scalability of the new algorithm is increased without reducing algorithm's performance. The experiment results also reveal that the local decisions made by the clients resulted in a higher local and global cache hit rates. Moreover, the memory overhead of the new algorithm is lower than that of the existing algorithms due to using of bloom filter data structure. Finally, the communication overhead is the lowest because of exchanging the small size cache summaries among the clients.

12. Bibliography

[1] Mursalin Akon, Towhidul Islam, Xuemin Shen and Ajit Singh (2010), "SPACE: A lightweight collaborative caching for clusters", Peer-to-Peer Networking and Applications, Vol.3, Iss.2, pp.83 – 99, ISSN: 19366442

[2] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder (2000), "Summary cache: a scalable wide-area web cache sharing protocol." IEEE/ACM Trans. Netw. 8, 3 (June 2000), 281-293. DOI=10.1109/90.851975 http://dx.doi.org/10.1109/90.851975

[3] Martin Kampe, Per Stenstrom, and Michel Dubois (2004), "Self-correcting LRU replacement policies." In Proceedings of the 1st conference on computing frontiers (CF '04). ACM, New York, NY, USA, 181-191. DOI=10.1145/977091.977117 http://doi.acm.org/10.1145/977091.977117

[4] Jeon, Won J., Nahrstedt, Klara (2004), "Cooperative caching for multimedia streaming in overlay networks." Department of Computer Science, University of Illinois at Urbana-Champaign, URL: http://hdl.handle.net/2142/10929

[5] Madhukar R. Korupolu and Michael Dahlin (2002), "Coordinated Placement and Replacement for Large-Scale Distributed Caches." IEEE Trans. on Knowl. and Data Eng. 14, 6 (November 2002), 1317-1329. DOI=10.1109/TKDE.2002.1047770 http://dx.doi.org/10.1109/TKDE.2002.1047770

[6] Gleb Skobeltsyn, Karl Aberer (2006), "Distributed cache table: efficient query-driven processing of multiterm queries in P2P networks." Tech rep LSIRRE-PORT-2006-010, EPFL, Lausanne, Switzerland

[7] Michael D. Dahlin, Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson (1994). "Cooperative caching: using remote client memory to improve file system performance." In Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation (OSDI '94). USENIX Association, Berkeley, CA, USA.

[8] Prasenjit Sarkar and John H. Hartman (2000), "Hint-based cooperative caching." ACM Trans. Comput. Syst. 18, 4 (November 2000), 387-419. DOI=10.1145/362670.362675 http://doi.acm.org/10.1145/362670.362675

[9] Francisco Matias Cuenca-Acuna and Thu D. Nguyen (2001), "Cooperative Caching Middleware for Cluster-Based Servers." In Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC '01). IEEE Computer Society, Washington, DC, USA.

[10] Purushottam Kulkarni, Prashant Shenoy, and Weibo Gong (2003), "Scalable techniques for memory-efficient CDN simulations." In Proceedings of the 12th international conference on World Wide Web (WWW '03). ACM, New York, NY, USA, http://doi.acm.org/10.1145/775152.775238

[11] Woo Hyun Ahn, Sang Ho Park, Daeyeon Park (2000), "Efficient cooperative caching for file systems in cluster-based Web servers" Cluster Computing, Proceedings. IEEE International Conference on , vol., no., pp.326-334, 2000, doi: 10.1109/CLUSTR.2000.889086, URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=889086&isnumber=19246

[12] Wikipedia (2011), "Cache", Web address: http://en.wikipedia.org/wiki/Cache

[13] Wikipedia (2011), "Memory hierarchy", Web address: http://en.wikipedia.org/wiki/Memory_hierarchy

[14] Wikipedia (2011), "Hash table", Web address: http://en.wikipedia.org/wiki/Hash_table

[15] Wikipedia (2011), "Memory locality", Web address: http://en.wikipedia.org/wiki/Memory_locality

[16] Microsoft (2011), "Caching in the Distributed Environment", Web address: http://msdn.microsoft.com/en-us/library/dd129907.aspx

[17] Java Ranch (2011), "Caching strategies", Web address: http://www.coderanch.com/how-to/java/CachingStrategies

[18] Buyya, Rajkumar et al; (2008), "Caching Techniques on CDN Simulated Frameworks," Lecture Notes in Electrical Engineering, Springer Berlin Heidelberg, URL: http://dx.doi.org/10.1007/978-3-540-77887-5_5 DOI: 10.1007/978-3-540-77887-5_5

[19] Wikipedia (2011), "Bloom filter", Web address: http://en.wikipedia.org/wiki/Bloom_filter

[20] OWChallie.com (2011), "Computer Organization & Architecture", Web address: http://www.owchallie.com/systems/cache-design-elements.php

[21] Dimitar, Popov et al. (2003), Cache memory implementation and design techniques, Web address: http://www.faculty.iu-bremen.de/birk/lectures/PC101-2003/07cache/cache%20memory.htm

[22] Wikipedia (2011), "Access time", Web address: http://en.wikipedia.org/wiki/Access_time