SKALA: Scalable Cooperative Caching Algorithm Based on Bloom Filters

Nodirjon Siddikov¹ and Dr. Hans-Peter Bischof²
¹ Enterprise IT Operation and Planning Unit, FE Coscom LLC, Tashkent, Uzbekistan
² Computer Science Department, Rochester Institute of Technology, Rochester, New York, USA

Abstract - This paper presents the design, implementation and evaluation of a novel cooperative caching algorithm, called SKALA, which is based on the bloom filter data structure. The algorithm is designed using decentralized architecture and offers highly scalable solution. It resolves the problems related to an inefficient use of the manager’s resources. The scalability of the algorithm is maintained by distributing the global cache information among cooperating clients. Furthermore, the memory overhead is decreased due to a bloom filter data structure which reduces the global cache size. The correctness of the algorithm is evaluated by running experiments. The experiment results demonstrate that SKALA decreases the manager load by the factor of nine which implies that it is more scalable compared to existing algorithms. The results also show a significant reduction in the memory overhead which implies that SKALA uses manager’s resources more efficiently.

Keywords: bloom filter, global and local cache, scalability, decentralized architecture, memory hierarchy

1 Introduction

The distributed systems, such as distributed file systems (DFS) and content distribution networks (CDN), usually employ caching mechanism. The primary goal of this mechanism is to temporarily store the frequently or recently accessed data, so that future data requests can be served faster. The benefit of caching can be extended by implementing a cooperative caching mechanism in these systems. Cooperative caching is implemented using two layers of memory – local and global cache. The distributed systems based on client/server architecture usually use the local cache to provide data reusability at client side. While the local cache is dedicated to specific client, the global cache is shared among all clients. The global cache’s primary goal is to help clients benefit from earlier accesses (to the same data) by other clients sharing the global cache. The content of the local and global caches are managed by special caching algorithms such as N-chance [4] and hint-based algorithms [5].

1.1 Research problems

The existing cooperative caching algorithms, such as N-chance and hint-based algorithms face several problems that prevent them from being applicable for wide variety of distributed systems. First, these algorithms do not scale when the number of clients or the local cache size is increased. The primary reason for such behavior is due to a centralized architecture implemented in these algorithms. This architecture uses single manager node which is responsible for the maintenance of the global cache. When the number of clients increases, the manager is required to handle more requests coming from clients and eventually becomes overloaded. Second, existing algorithms create a memory overhead on the manager side. As the number of clients increase, the global cache size at the manager also increases gradually which causes a memory overhead.

1.2 Proposed solution

This paper addresses the research problems by introducing a novel cooperative caching algorithm, called SKALA, which is based on a bloom filter data structure. SKALA uses decentralized architecture that relieves the manager from global cache maintenance and distributes the global cache among client nodes. Thus, clients would have their own copy of the global cache so that they avoid contacting the manager for the data. This feature reduces the load on the manager to maintain the scalability of the algorithm. Moreover, SKALA decreases memory overhead by incorporating the bloom filter into the global cache. The proposed solution is suitable for distributed systems that operate in client/server environment.

In order to evaluate the performance of the algorithm, a trace-driven software simulator has been developed. The experiment results demonstrate the effectiveness and correctness of the algorithm. According to the results, SKALA performs better compared to the existing algorithms when the number of clients or the local cache size increases. In particular, the bloom filter reduces the global cache overhead by factor of nine and maintains the algorithm’s scalability over many client nodes.
1.3 Motivation

The primary motivation of this paper is to measure the scalability of the cooperative caching algorithms. Another motivation is to explore the relationship between cryptography and caching algorithms. The bloom filter data structure, for example, establishes this relationship, because it uses a cryptographic hash function to insert, lookup and delete its elements. Moreover, it was also claimed that bloom filter can be applied for caching algorithms to save memory space [2]. The bloom filters and hash functions are relatively new in the current research area, thus it is interesting to see how they help cooperative caching algorithms be scalable.

1.4 Organization of paper

The current paper is organized as follows: Section 2 describes the required background knowledge and concepts. Section 3 elaborates more into existing cooperative caching algorithms. Section 4 examines the research problem and introduces the novel caching algorithm. Section 5 describes the experiments and the metrics used to evaluate the algorithms. The limitations of this paper and the future works in this research field are discussed in Sections 6 and 7, respectively. Finally, Section 8 summarizes the paper by presenting the conclusions drawn from the experiment results.

2 Background

2.1 Cache memory

A cache memory transparently stores the data so that future data requests can be served faster [9]. It can be used to increase the performance of secondary storage devices, such as a hard disk, by saving the copies of frequently or recently accessed blocks. 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 data stored in the cache may either be a previously computed value or the block that has been requested before.

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. In such setting, cache filters out the data request directed toward the slower layers of the hierarchy. For example, when the client requests the data block, it first checks the cache. If the requested data 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 data has to be fetched from the slower memory layer such as a disk. Therefore, the performance of the system will be improved if more data is read from the cache [9]. The size of a cache depends on the requirements of the system, but it is usually kept small to maintain its efficiency.

2.2 Cooperative caching system

A typical cooperative caching system in client/server environment usually implements the local and global caches. The local cache is implemented in the client side as part of its memory hierarchy. The global cache is also considered as a layer in the hierarchy and it is positioned between a local cache and the server’s storage [5]. Its content is managed by a cooperative caching algorithm using the logic of least recently used (LRU) and least frequently used (LFU) algorithms.

The architecture of cooperative caching system usually involves a manager, clients and a server. The manager implements and maintains the global cache; the server hosts the data blocks and the client sends the read requests to these blocks. The manager is responsible for the coordination of the clients’ requests to the blocks. The extent of coordination is different among various cooperative caching algorithms and it is considered a major distinction between them [5]. Moreover, the clients have to cooperate with each other when accessing the blocks. Cooperation takes place when one client does not have the required block in its local cache and it probes caches of other clients for the required data.

Cooperative caching systems are usually designed using a centralized architecture. In this architecture, if client experiences local cache miss, then it contacts the manager to lookup the global cache. The manager usually knows which client’s cache contains the required block through its global cache. 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 an expensive process. Figure 1 illustrates an example of such centralized architecture.

Figure 1: A centralized cooperative caching system. In this system, a single manager is responsible to maintain the global cache.
3 Existing algorithms

3.1 N-chance algorithm

The N-chance algorithm is one of the pioneers in the field of cooperative caching. It uses a centralized architecture which includes clients, the manager and the server. In this algorithm, the local cache is located in the clients’ side. But the global cache is maintained by the manager and has the comprehensive view of the all local caches. Local cache content is managed by LRU cache replacement algorithm, but the global cache is coordinated using a more sophisticated policy. In this policy, the data block is either discarded or kept in the global cache based on its quantity in the cooperative caching system. If there is only one specific block in entire system, it is called a singlet; otherwise it is considered as a non-singlet. The singlet data blocks are forwarded to a randomly selected client to further retain them in the system; otherwise, they are discarded immediately. Each singlet block is given a recirculation count N such that every time the block is forwarded, the count is decremented. In other words, the block is given N chances to stay in the cooperative caching system. When the count reaches zero, the block is discarded regardless of its singlet status [4].

The central manager component plays an important role in the N-chance algorithm. It is responsible for global cache maintenance and lookup, and forwarding the block requests to the clients and the server. Every time client inserts or removes a block from its cache, it reports these changes to the manager. The manager uses this information about changes to maintain the consistency of the global cache [4].

3.2 Hint-based algorithm

In N-chance algorithm, the information contained in the global cache is considered as facts. This is because global cache reflects the factual data stored in the local caches of clients. However, maintenance of these factual data increases overhead on the manager. The hint-based cooperative caching algorithm attempts to solve this overhead problem by relaxing 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. Thus, the global cache is maintained by both clients and the manager [5]. In such setting, the manager still retains the control of the global cache, but it incurs much less overhead from clients.

Facts and hints are main components of the hint-based algorithm. They usually contain the information about the location of a master block, which is the original block fetched from the server. The hints reduce the client’s dependence on the manager when performing caching operations. However, the hints are not always accurate, because they 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. However, managing hints is less costly for a client than managing facts, because the accuracy of hints needs not to be maintained at all times [5].

4 New algorithm: SKALA

4.1 Problems with existing algorithms

There are several problems associated with existing cooperative caching algorithms. The first problem is related to their scalability – the algorithms do not scale when the number of clients increase in the cooperative caching system. Increased number of clients generate excessive amount of block requests towards the manager and causes it to get 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 which causes a communication overhead among the clients and the manager. In the case of hint-based algorithm, the local hints may become obsolete, thus clients will be required to contact the manager to obtain the fact. Such algorithm also produces a communication overhead as the number of clients increases. Also, it may take a while until hints are updated, so there is a risk for a client to perform an inaccurate caching operation based on the obsolete hint. The outcome of this operation is unfavorable since inaccurate decisions increase the cache miss rate [8].

Another problem of existing algorithms is associated with a memory overhead placed on the global cache. As more clients are added to the system, the manager has to allocate more memory space for the global cache. While the hint-based algorithm mitigates this problem by using hints, it still maintains facts in the global cache at the cost of manager’s resource. This may result in a waste of memory resource when the number of obsolete facts becomes excessive.

4.2 Proposed solution

In order to overcome the problems with existing algorithms, current research paper proposes a solution to reorganize the components presented in Figure 1. In this solution, the global cache component is removed from the manager and distributed among the clients. Thus, each client maintains its own copy of the global cache besides its local cache. A client periodically updates the state of its global cache. Updating procedure involves calculation of bloom filter from local cache and broadcasting it among the other clients. Figure 2 shows the overall architecture of the proposed solution.
example, the lookup operation may result in false positives and negatives. The false negative happens when the particular block is stored in the cache, but the bloom filter says otherwise. On the other hand, the false positive occurs when the block is not stored in the client’s cache, but the bloom filter says otherwise. Even though these errors affect the cache hit ratio, bloom filter keeps the logic of the cooperative caching algorithm intact. For example, a false cache hit does not result in a wrong block being served [2]. The probability of getting false positives and negatives can be controlled by proper choice of values for bloom filter parameters (such as its size and expected number of elements). One of the advantages of bloom filter is its fixed size so that the capacity of the global cache does not change over time. On the other hand, the bloom filter can only be used for cache lookup operation, and it cannot directly be used to store the block’s other attributes such as its size [7].

Figure 2: The architecture of the proposed solution. In this architecture, the global cache component is removed from the manager and distributed to each client.

The proposed solution is implemented by a novel cooperative caching algorithm called SKALA. It is designed using decentralized architecture and requires each client to maintain local and global cache components. The local cache stores the accurate state of the data block including its content. But, the global cache stores only approximate information about blocks. Due to such approximate information, the global cache lookup operation may return false positive (i.e. false cache hit) and false negative (i.e. false caches miss) results. However, in SKALA, the local cache is accessed more frequently than the global cache. Therefore, the algorithm does not keep the global cache up-to-date at all times and false positives and negatives happen occasionally within tolerable threshold. SKALA also reduces the global cache size by using bloom filter. This assures that global cache uses less space in client’s memory which increases the local cache hit rate.

4.3 Data structures used in SKALA

SKALA uses traditional and novel data structures to implement the cache memory (Figure 3). The local cache is implemented using a hash table which keeps each data block as a key/value pair. The key represents the block’s ID and the value denotes the block’s content. Unlike a hash table, a bloom filter only stores the block’s ID as a key, but the block contents are not stored [6]. Such design gives an advantage during caching operations. For example, a bloom filter spends less time to look up the block information compared to the hash table. But, the time to access the block contents stored in the local cache becomes significant because of the network latency [1].

The bloom filter data structure is considered as an exceptional feature of the SKALA. It is a probabilistic data structure which basically is an array. Similar to an array, it supports insert, delete and lookup operations. However, the result of the lookup operation is returned with some level of probability. For
5 Evaluation

5.1 Evaluation metrics

The following metrics are used to evaluate the performance of the cooperative caching algorithms:
- **Manager load** measures the overhead a client imposes on the manager. The overhead is expressed as a number of messages generated by the client to communicate with the manager. This metric is important because it represents the amount of work completed by the manager for cooperative caching operations.
- **Memory overhead** measures the overhead imposed on the manager’s memory. The size of the global cache is used to determine the memory overhead. Such measurement is also an important factor in estimating the algorithm’s scalability.

5.2 Experiment results

In order to conduct the experiments, a special software simulator is developed in Java which implements all cooperative caching algorithms under investigation. The simulator used a single Java Virtual Machine and its input parameters are adjusted through a configuration file. List of algorithms tested are provided below:
- N-chance algorithm;
- Hint-based algorithm;
- Non-bloom filter based algorithm;
- Bloom filter based algorithm (SKALA).

The input parameters of the simulator specific to caching algorithms (local cache size, the number of clients and the network latency) are adjusted regularly based on the requirement of the experiment.

5.2.1 Manager load

Figure 4 shows the measurement of the load imposed on the manager. The load is directly proportional to the amount of messages exchanged between clients and the manager. These messages are broken down to consistency, replacement and lookup messages. According to the figure, the N-chance algorithm imposes most load on the manager compared to the rest of the algorithms. Its centralized architecture requires clients to make frequent contacts to the manager to lookup the global cache. Another reason is because a client constantly updates the manager with the change in its local cache content for the purpose of global cache maintenance. Consistency messages put most of the load on the manager due to their importance for the global cache. In the case of the hint-based algorithm, the manager load is three times less than that of N-chance. Despite using hints, the algorithm still requires clients to contact the manager in order to maintain the facts in the global cache. Finally, the decentralized architecture of SKALA imposes fewer loads on the manager making the algorithm scalable. Because the global cache is locally stored in each client, it does not contact the manager at all for lookup, insert and delete operations.

**Figure 4:** The average load imposed on the manager by each algorithm. The load is defined as the number of messages sent and received by the manager. The manager load is broken down to consistency, replacement and lookup messages.

5.2.2 Memory overhead

Figure 6 shows the result of the experiment which evaluates the memory overhead of the algorithms. This research used the
value of the global cache size to estimate the memory overhead. This value is variable and depends on the aggregate sizes of local caches. According to figure 6, the memory overhead of SKALA is significantly low compared to that of other algorithms. SKALA uses a bloom filter to implement the global cache which keeps its size fixed regardless of number of blocks. On the other hand, the memory overhead is the highest in case of non-bloom filter based algorithm. Such huge difference is due to the method used to store the blocks in the global cache. This algorithm stores the exact copy of each block in the global cache. However, the rest of the algorithms use block’s ID to populate the global cache and the actual block contents are stored in the local cache.

Another experiment evaluates the scalability of the algorithms when the local cache size varies between 64 – 1024 KB. The experiment assumes that the size of the global cache is proportional to size of the local caches in all algorithms. The result of experiment is shown in figure 7. In case of the non-bloom filter based algorithm, the global cache size increases exponentially as local cache size reaches the 1024 KB mark. This algorithm stores the block contents in the global cache, thus local cache content consumes more space in the global cache. N-chance algorithm shows similar behavior but at lesser extent due to allocation of global cache maintenance to the manager. The hint-based algorithm achieves the better result than N-chance through keeping merely master block information in the global cache. Finally, SKALA demonstrates immunity to the increase in the local cache size due to the fixed size of the bloom filter used in its global cache.

The final experiment uses the number of clients to evaluate the scalability of the algorithms (Figure 8). According to the figure 8, the number of clients in the system directly affects the memory overhead. N-chance and hint-based algorithms demonstrate that the overhead on the memory grows exponentially when number of clients increase. Such exponential growth indicates that these algorithms are not scalable due to increased block access time and increased load on the manager. Moreover, both of these algorithms use a variable size global cache which can grow infinitely. Thus, when the number of clients increases, the amount of cached block information consumes more space in the global cache. On the other hand, SKALA’s global cache size remains relatively stable until the number of clients reaches 50. The global cache allocates one bloom filter for each client’s local cache information, thus exceeding this threshold of 50 clients is sufficient for to increase the size of the global cache.

6 Limitations of proposed solution

The proposed solution faces some limitations that could affect its performance. For example, the bloom filter data structure only stores the approximate state of the global cache. This may result in false positives and negatives which means not all global cache lookup operations return accurate answer. Inaccurate answers are primary reasons for the local and global cache misses. Consequently, this increases the block access time and reduces the performance of SKALA. Therefore, the bloom filter has to be created with appropriate input parameters. Moreover, SKALA does not implement the cache
writing policy. However, integrating this policy into SKALA would make the contents of cache memory and server disk more consistent. On the other hand, such modification may not change the outcomes of the experiments, because cache writing and block reading policies are mutually exclusive and they do not affect the cache hit/miss rates. Another shortcoming of SKALA is associated with the input data used in the experiments. The input data (trace) is synthetic meaning that it 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.

7 Future work

Even though SKALA proved itself as scalable caching algorithm, there are some enhancements that could be applied for the algorithm. One of them entails implementing a cache writing policy in SKALA. According to this policy, when the content of the cache in SKALA 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 SKALA is to implement an offline caching algorithm. All of the algorithms presented in this paper are all considered online algorithms. The offline algorithm can be used in the experiments to measure the upper performance bound of these online algorithms. Such measurement would help optimize the performance of the existing algorithms. Finally, implementing a multithreading would improve the performance of SKALA. For example, the computation of the bloom filter can be parallelized with multithreading so that each hash functions are calculated by separate thread.

8 Conclusion

A cooperative caching system usually implements two layers of cache memory: local and global caches. The system components usually consist of the manager, the clients and the server all of which works under the rule of the specific caching algorithm. The algorithm is also used to coordinate the contents of the local and global caches. The level of coordination is an important factor and is the main difference between various cooperative caching algorithms. The current paper introduces the bloom filter based decentralized caching algorithm, called SKALA, which focuses on the scalability and efficient use of global cache memory. SKALA achieves the scalability by removing the maintenance of the global cache from the manager. Instead, it distributes the entire global cache among the clients. The content of the global cache is composed of a set of bloom filters, one filter to represent each client’s local cache. The bloom filter simplifies the implementation of the global cache replacement policy in SKALA which does not involve the manager. The experiments on SKALA show that the load on the manager is decreased due to algorithm’s decentralized architecture. This assures the scalability of SKALA without reducing its performance. Moreover, the bloom filter reduces the memory overhead for the global cache. This signifies that SKALA is better at using the manager’s resources efficiently compared existing solutions.

9 References