Replacement Algorithms in Cooperative Caching

Manu Menon
mxm1752@rit.edu
M.S. Project Final Report
Advisor: Prof. Hans-Peter Bischof
August, 2016
Contents
List of Figures
1 Introduction

1.1 Cooperative Caching

In a distributed system where clients make requests for data from a server, clients may cache frequently or recently used data locally in order that these may be read from the local cache when required rather than making another request to the server over the network. Local caching can thus improve the performance of the system over the case where no local caching takes place and all requests are fulfilled from the server. However, in a distributed system with the involvement of several clients, it is possible that a block requested by a client that is not contained within its local cache may be contained within a peer’s local cache. Therefore rather than requesting the block from the server, serving the block from the local cache of the peer on the network may be less costly. This is the basic idea behind cooperative caching. Clients cooperate to share their caches so that local cache misses may be fulfilled from the share cache, with the server fulfilling the request only if neither the local nor the shared cache are able to do so. The shared cache is typically centrally coordinated by a designated master. Requests that cannot be fulfilled from the local cache are forwarded to the master to see if it may be fulfilled from the shared cache. If not, the request is forwarded to the server.

![Master coordinated shared cache](image)

Figure 1: Master coordinated shared cache
1.2 Replacement Algorithms

Two important questions arise in the operation of a cooperative caching system. When the local cache cannot accommodate an incoming block, a locally cached block must be replaced. This is known as the algorithm’s local replacement policy. The second question regards how to handle the displaced block. Most techniques attempt to keep high value blocks in circulation, and different heuristics may be used to determine which blocks are of greater value than others. Some algorithms, in addition to a local replacement policy, also specify a global replacement policy which determines which block in the global cache, if any, the displaced block will replace if it is determined that the displaced block is of high enough value per the employed heuristic to be preserved rather than discarded.

2 Problem Statement

A common feature of several well known replacement algorithms (discussed elsewhere in this report) is the use of a local replacement policy based on recency of use. This is typically accomplished by using an LRU cache. The LRU cache maintains cached items in a linked data structure where new and last read items are moved to the head of the list. In the process of mov-
ing this item to the head a cached item may be displaced if the cache was already at capacity. This displaced block is handled according to the algorithm’s policy regarding such blocks. While recency of use is a useful heuristic to structure local caches and replacement policy, recency of use does not always coincide with popularity since the cache preserves recently read blocks, which may not be the most frequently read blocks over the period of network activity. The present work investigates a cache replacement policy based on frequency or popularity of use, using the number of clients caching a block as a working heuristic for popularity. Three variations of this approach were implemented and compared with the performance of standard algorithms using several measures. Additionally a fourth experimental algorithm tailored specifically for the case of requests that consist of known sequences of blocks was also implemented and studied. The procedure and results of this study are presented here.

3 Existing Work

A large body of work exists in the field of cooperative caching. Pioneering work on the subject was done by Dahlin et al.\cite{Dahlin1972} and two out of the three standard replacement algorithms studied in this work were described in that work. In addition to the work implemented here some different approaches to cooperative caching replacement were studied but were not implemented as part of this work. These include Cooperative LRU \cite{Hamdi2000} which moves a cached block closer to the tail of an LRU cache as the number of copies in the shared cache increases, and hint based approaches \cite{Cao2003} in which clients maintain local hints regarding the potential location of a block in the shared cache. Several variations on the basic LRU concept such as weighted LRU and LRU Min \cite{Liu1994} were considered as a starting point for this research. This work implements three standard algorithms for cache replacement in a cooperative caching system. These algorithms are described in this section.

3.1 Singlets

A singlet refers to a block cached at a single client. In some collaborative caching approaches, singlets are considered to be uniquely valuable. If a singlet is discarded, a subsequent request would result in a disk read. In order to avoid this relatively expensive operation, a focus of collaborative
caching strategies is the handling of singlets. It is important to note here that
the production of singlets is also of unique importance in the collaborative
caching area. The reason for this is easy to fathom. A block that is duplicated
in multiple clients’ caches takes up positions that may be occupied by other
disk blocks currently not in any client’s cache. Replacement algorithms are
therefore usually fairly aggressive in discarding locally cached blocks that are
also present in other client caches. This encourages the production of singlets
during replacement. On the other hand, a block read from the shared cache
by a client is usually cached locally by the requesting client. This obviously
creates duplication, undermining the production of singlets. However, since
a local cache hit is much less expensive than a collaborative cache hit, most
caching strategies preserve this local cache policy. If one now considers a
scenario where for example a collaborative cache hit is very close in cost
to a local cache hit, then it is a conceivable strategy for clients to cache
only blocks read from disk and use the shared cache for everything else. This
results in essentially no duplication across the cache thereby favoring singlets
very aggressively. Such a strategy maximizes singlets and by this strategy the
collaborative cache can hold the greatest number of disk blocks. A drawback
of such a policy is that since the portion of a clients cache that is pooled can
only be accessed when that client is idle, a local cache hit may turn out in
practice to be much less costly than reading from the shared cache.[?] 3.2 Greedy Forwarding

The greedy forwarding algorithm is in a sense the basic co-operative
caching algorithm in that it treats peer caches as an extension of the client’s
local cache. Requests that are not fulfilled from the local cache are then for-
warded to the global cache and blocks requested by any client may be cached
at any other client (the greedy forwarding part) so long as that client is idle
and is able to cache the block. After the warm up phase active clients can
no longer cache blocks on behalf of other clients, so the algorithm reverts to
local least recency of use replacement through a standard LRU cache, and re-
quest fulfillment through the shared cache.[?] The master in this algorithm is
limited to indicating which client is caching a given block so that the request
may be forwarded appropriately. Greedy forwarding makes no distinction
in the handling of singlets or any other type of block. No forwarding takes
place so singlets are discarded based on recency of use.

4
3.3 N-Chance

The N-Chance algorithm relies on a simple policy for the handling of singlets. A singlet is forwarded to a randomly selected client. A given block may be subjected to such re-circulation a predetermined number of times, referred to as the re-circulation count, before it is discarded. The forwarded block reaches the receiving client where it may displace a block in the receiver's cache. To prevent endless re-circulation, this secondarily displaced block is not subject to forwarding but simply discarded. The re-circulation count of the forwarded block is decremented in the process. The standard N-Chance implementation uses an LRU cache, implying that the block subject to replacement is the least recently used block in the client's local cache. The least recently used block is ejected from the cache upon caching the incoming block and then an inquiry is initiated to the master to determine how many clients are caching the replaced block. If the client initiating this request is the only client caching the block, the block is forwarded as described above. The advantage of the N-Chance algorithm is that it proposes an extremely simple strategy for keeping singlets around and available to clients. It does not introduce any additional computational overhead on the master besides the relatively simple request to forward a block to a random client. On the other hand, the random forwarding and subsequent replacement may be sub-optimal in the sense that the subsequent replacement may also replace a singlet which has no opportunity for forwarding.
3.4 Robin Hood

The Robin Hood algorithm for replacement in cooperative caching seeks to improve on the N-Chance algorithm by forwarding singlets in a targeted manner.[?] Through this targeted forwarding the robin hood algorithm tries to overcome the possible shortcomings of random forwarding as prescribed by the N-Chance algorithm. The block selected for forwarding is, as in the N-Chance algorithm, determined by the use of an LRU cache. The least recently used block in the local cache is removed from the cache when the incoming block enters the cache. The master is then queried to determine how many clients are caching the displaced block. If multiple clients cache the block, it is simply discarded. If it turns out that the local cache was the only one holding the block (a singlet) the master is queried for the global block to be replaced. The block to be replaced is determined by the master to be the block that is cached by the greatest number of clients. So the most cached block is replaced by a block cached at only one location, hence the name Robin Hood, a reference to the replacement of “rich” (cached multiple times) blocks with “poor” (singlet) blocks. The requirements of the robin hood algorithm imply that the master must not only maintain a count of how many clients are caching each block (as it must do in the N-Chance algorithm as well for the determination of which blocks are singlets), but it must also maintain this count in sorted order, in order to be able to supply the victim client and block as they are called, for replacement. This introduces a
significant computational overhead on the master which must be taken into account when evaluating the algorithm’s performance.

4 Proposed Study

In addition to the 3 algorithms above which are standard algorithms in the field of cooperative caching, and have been studied in detail elsewhere, 3 experimental algorithms were implemented for this project. These experimental algorithms eschew the LRU cache in favor of a simple cache. Local cache replacements as well as forwarding are handled by the master. The master accomplishes the local replacement and targeted forwarding where applicable using the number of clients caching the block in question as the guiding heuristic. In order to make this determination, the master has to maintain a sorted order over the shared cache so that the most cached block in the cache and the clients caching that block may be provided upon request. This additional overhead on the master is similar to that imposed by the Robin Hood algorithm. A fourth experimental algorithm was implemented that is designed specifically to take advantage of cases where the data from the server comprises known sequences. This algorithm is known as K-Sequence. These four experimental algorithms are described below.

4.1 No Forwarding

This algorithm is similar to the greedy forwarding algorithm meaning that singlets are not forwarded if they are displaced during caching. When a block is read from disk or shared cache and a locally cached block is to be replaced the block that is cached by the largest number of clients is selected for replacement. The idea behind this algorithm is that the targeting of local blocks that are cached at multiple locations should result in the rapid production of singlets. The algorithm does not include any forwarding of these singlets so they are simply discarded if encountered during local replacement. The reason for this is that since the most cached local block is chosen for replacement, if that block is a singlet, there is no other local block that is cached at more than one location, since if such existed it would have been selected for replacement. Since all clients follow such a replacement policy, all client caches should tend toward singlets as the simulation proceeds.
4.2 Random Forwarding

This algorithm shares the local cache replacement logic with the no forwarding algorithm, so the local block targeted for replacement is the block that is cached by the greatest number of clients. The handling of displaced singlets is where the 2 algorithms differ. The random forwarding algorithm treats displaced singlets in the same manner as the N-Chance algorithm. Singlets are forwarded to a client selected at random by the master. The re-circulation count is decremented and secondary forwarding is prevented. So if the forwarded singlet arrives at the randomly selected client and happens to displace another singlet when it is cached, then this second singlet is simply discarded. This algorithm should show an improvement over the no forwarding algorithm since singlets are kept around longer by recirculating them.

4.3 Targeted Forwarding

This is another variation on the no forwarding algorithm and shares the same local replacement logic. While the locally cached block selected for replacement is the block that is cached by the greatest number of clients, the handling of displaced singlets in this algorithm is similar to the Robin Hood algorithm. This means that during singlet forwarding, the master selects a victim client and a victim block. The victim block is the block in the shared cache that is cached by the greatest number of clients. The victim client may be any client caching the victim block except the forwarding client. Once the master has selected the victim block and client, the singlet is forwarded to the victim client where it replaces the victim block. The re-circulation count is decremented and no secondary forwarding is permitted. The targeted forwarding algorithm is expected to show a further improvement over the random forwarding algorithm by selecting a heavily cached block for replacement.

4.4 K-Sequence

The K-sequence algorithm is designed to provide an effective policy for a collaborative cache in the following case. Consider a scenario where blocks are requested by clients in a known sequence. In other words if block A is requested, then it is guaranteed that a sequence of k blocks, say B through
E will be requested, and this is true for all clients in the network. The K-Sequence algorithm is designed for this special case of request sequences, and essentially treats a sequence in the same way that the algorithms treat a single block for the purposes of caching and replacement. When the cache cannot accommodate the incoming sequence without replacement, then the entire sequence is forwarded to a random client as a block would be in the N-Chance algorithm. The K-Sequence is expected to perform better than the other algorithms in the case described above.

5 Simulation

A network simulation was implemented in Java in order to run the different cache replacement algorithms studied here. The simulation utilized the Java 1.8 JDK. The structure of the objects in the simulation is as follows. For the purposes of this simulation, cache size is represented in blocks and network events are counted in ticks. This means that the simulation does not consider for example file size or the size of individual blocks, rather a cache size in blocks is used. Client objects are thin pojos, implement the Callable interface and own a cache implementation. This could be either a simple cache, an LRU cache, or a K-LRU cache, a variation of LRU cache used for the K-Sequence implementation. Each client thus runs in its own thread. The entire simulation is run within a thread pool and the pool is shut down once all futures are received. The replacement algorithm classes implement the Algorithm interface, and every client has a reference to an instance of the Algorithm implementation. Since all the algorithms under study use a single master, the Master classes are implemented as essentially static classes. There are 2 types of Master class, one implementation maintains a representation of the shared cache sorted in descending order of the number of clients caching a block, the other implementation also maintains a representation of the shared cache in the form of a map determining which clients cache a given block, but no sorted structures are required or maintained. A configuration file determines the initial state of the simulation: number of clients, cache size, disk size, total requests etc. The configuration also determines the trace generator to be used. One of 3 such generators may be used. The first simply generates random sequences within the parameters of the number of blocks on disk and total requests. The second generator produces less random sequences that include many contiguous sequences of
blocks. The third generator is used to generate traces to be processed most effectively using the K-Sequence algorithm although these can be used for any of the implemented algorithms.

6 Results

The first experiment measured the performance of the different caching algorithms as the cache size was increased. This experiment was conducted for different network sizes varying from 50 to 150 clients. The results for the 50 client simulation are shown in the figure below. The results for the experiments with larger numbers of clients were comparable. For the

![Figure 5: Effect of variation in cache size - cache hits](image)

Figure 5: Effect of variation in cache size - cache hits
standard algorithms, Robin Hood shows a consistently better hit rate than N-Chance which itself shows a better cache hit performance than Greedy Forwarding. Of importance to note is that as the cache size increases past the point where duplication across the cache is minimized, Robin Hood converges back towards N-Chance performance. This is noted in previous work on this algorithm. The experimental algorithms all show very similar performance although targeted forwarding is very slightly better than random, itself very slightly better than no forwarding. These results although small in difference were, significantly, consistent across all trials.

![Figure 6: Effect of variation in cache size - cache misses](image)

The effect on cache miss rate is also shown above to be as expected from the previous graph. Also notable here is the convergence of Robin Hood to
N-Chance performance as duplication across the cache reduces which also reduces the effectiveness of targeted forwarding approaches which become no better than singlet forwarding if duplication is minimal.

The second experiment measured the performance of the different caching algorithms as the cache size was held constant and the number of clients was increased. The number of clients in each run was increased by an increment of one upto a total increase of 30 clients. This was rerun in various intervals from 5 to 35 clients, 10 to 40 and so on upto a maximum of 100 to 130 clients. The number of blocks on disk was proportionately increased for each run.

![Figure 7: Effect of variation in number of clients - cache hits](image)

In this experiment, as in the previous one the relative performance of the standard algorithms shows Robin Hood to have the highest cache hit
rate with N-Chance second and Greedy Forwarding in last place. The convergence of Robin Hood to N-Chance performance as the number of clients increases, which results in reduced duplication across the cache, is clearly visible here. The experimental algorithms once again show only a slight variation in their relative performance, but this was consistently reproduced across all trials. Targeted forwarding had a slight edge on random forwarding which was slightly better than no forwarding.

Figure 8: Effect of variation in number of clients - cache hits

The cache misses diagram is in agreement with the previous results with respect to both the standard as well as the experimental algorithms. All 4 algorithms using targeting start out with roughly the same cache hit/miss rate. As duplication across the cache reduces due to more clients joining,
the LRU cache based algorithm (Robin Hood) converges towards N-Chance while the targeted local replacement policy algorithms continue to perform better, but with small (relative to each other) benefit from their respective global replacement policies. These results are further discussed below.

![Figure 9: K-Sequence varying cache size - cache hits](image)

The third experiment involved measuring the performance of the K-Sequence algorithm as compared to some of the standard algorithms implemented by varying cache size. The particular implementation used for the K-Sequence in this study was using an LRU cache and random forwarding similar to the N-Chance algorithm. Therefore the most meaningful relatively study was to compare the performance of K-Sequence with those of
N-Chance and Robin Hood, since both use the LRU cache, N-Chance forwards randomly similar to the K-Sequence implementation and Robin Hood uses targeted forwarding. The K-Sequence trace generator was used with all 3 algorithms for these experiments. The result of the experiment showed that K-Sequence performed better than N-Chance for cache hits/misses but not as well as Robin Hood.

The fourth experiment measured the performance of K-Sequence compared to N-Chance and Robin Hood when the number of clients is varied. Here also the results were similar to the previous experiment. K-Sequence showed better cache hit rate than N-Chance but worse than Robin Hood with all 3 algorithms using the K-Sequence traces and the number of clients increasing in increments of 10 from 40 through 120.

Figure 10: K-Sequence varying cache size - cache misses
Figure 11: K-Sequence varying total clients - cache hits
Figure 12: K-Sequence varying total clients - cache misses
7 Conclusions

The findings of the study are as follows:

1. For the standard algorithms run under various conditions and variations the Robin Hood algorithm performed better by all measures used than N-Chance algorithm which in turn was better than the Greedy Forwarding algorithm.

2. For the experimental algorithms the Targeted Forwarding algorithm performed better than the Random Forwarding algorithm which in turn was better than the No Forwarding algorithm for all measures used. However the difference in their results, although consistent across the experiments, was extremely small as a percentage.

3. Algorithms where the master is required to maintain the shared cache representation as a sorted structure performed significantly slower than others in this simulation. However when measured by total cache hits or total disk reads, these algorithms performed the best. An obvious implication arising out of this result would be that it is likely that at certain levels of load on the master, the benefits of targeted algorithms are superseded by the additional computational load on the master. Since the simulation assigns tick counts to network events, and doesn’t specifically measure the overhead on the master in maintaining the shared cache in required order, this determination was beyond the scope of this study, but remains an interesting avenue of possible future work. The referenced paper on the Robin Hood algorithm acknowledges this overhead on the master but does not consider this question.

4. For the algorithms that use the targeting approach, the difference between the three algorithms was consistent but in terms of percentage, very small. The conclusion drawn from this is that the heaviest contribution to the production of singlets comes from targeted local replacement, which is the fundamental replacement policy of all three algorithms. In other words if a local replacement policy based on block richness is used, the global replacement and/or forwarding policy makes a very small additional difference. All three performed significantly better than the standard algorithms in all experiments.
5. The K-Sequence algorithm was found to produce good hit rates compared with N-Chance. Since this K-Sequence implementation and the N-Chance algorithm both use an LRU cache, this is a significant result using the same trace generator. Although the K-Sequence algorithm performed worse than Robin Hood which uses targeted forwarding, if the implementation were altered to use targeted local replacement, targeted global replacement or both, a more favorable results with respect to Robin Hood may be obtained.

8 Future Work

The study produced some interesting results which could be taken further in future research. The clear advantage of the targeted local cache replacement policy may be offset as load on the master increases. This gives rise to two areas of future work. One would be to investigate how the master in such cases may be made more efficient and how to alleviate as far as possible the load on the master. Possible techniques that can be used here could be periodic sorting of the cache instead of maintaining sorted order through every addition and removal. Ultimately this becomes an issue of scalability, so the second related area of work could be to try to establish the relative scalability of targeted and random forwarding approaches to determine under what circumstances any advantage targeted approaches may have is neutralized.

As observed in this study, the Robin Hood algorithm starts to perform similarly to the N-Chance algorithm once either the number of clients or the cache size becomes large enough that duplication of blocks across the shared cache (and therefore the advantage of targeted replacement) becomes minimal. A related point of research would be to alleviate master overhead by switching from Robin Hood to N-Chance once the head of the sorted cache is located at only one client, and back to Robin Hood if this condition changes. Such automatic selection of replacement algorithm based on certain conditions could be a fruitful line of future work.

The K-Sequence algorithm shows promise when used with trace files designed to be a good fit for the premise of the algorithm. This implementation used an LRU cache and random singlet forwarding, similar to N-Chance and performed slightly better than N-Chance in all experiments, although not as well as Robin Hood. A line of future work would be to modify the implementation to use targeted local replacement, targeted shared replacement
or both and compare those results with the Robin Hood and other targeted algorithms to see if an improvement over them is possible.

An investigation of cooperative caching applications in private networks such as content delivery networks is another area of possible future work. CDNs such as Akamai deliver enormous amounts of content through global networks. Extension of the simulation described here to clusters of clients each under the coordination of a master could be a good way to approximate the behavior of such networks.\[?\]

Much work has been done on investigating the application of probabilistic data structures such as Bloom filters to cooperative caching. The algorithms outlined here could be studied against those using such data structures to determine their relative merits and applicability.