Analysis of Trace and Cache Blocks in Distributed Co-Operative Caching

Anonymous
Department of Computer Science
Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, NY 14586

I. PROBLEM STATEMENT

This paper describes the working of existing caching algorithms for distributed co-operative caching, namely Greedy algorithm, N-Chance algorithm and Round Robin algorithm. The comparison of each of these algorithms are documented to see which algorithm works best. We also analyze the trace or request patterns from each client. To improve the system efficiency of the system, the prediction of the request by clients enables us to pre-fetch the data in the clients’ cache.

II. INTRODUCTION

A. Caching

Caching is a process of storing data that are frequently used by a client/node. This ensures faster access of data or memory block and this drastically minimizing the server/disk access time. This is because the cache copy is stored in a high-speed memory. Cache memory access time, is 20-25% lesser than the time needed to access main memory [1]. Caching has also shown to decrease the bandwidth consumption. There are various areas in which caching is used like networking, mobile computing, web caching, etc. One such is the co-operative caching in distributed networking. Our goal in this paper is to compare and study different collaborative/co-operative caching algorithms.

B. Collaborative Caching

Distributed collaborative caching motivates the idea of using peer client caches as a global cache system. Collaborative caching is needed because the processor speed is increasing exponentially when compared to the disk performance. The processor acts on a given task faster than server, thus the bottleneck of every system is the disk access time. To increase the performance of any distributed system, caching a memory block defeats the limitation of lesser disk access speed [2]. Thus, larger the cache size, due to the global cache, faster is the performance to handle data requests. The Fig.1 represents how a client uses peer’s client as its own.

C. Prefetching

In this paper, we propose prefetching schemes to work along the collaborative caching algorithm. The prefetching is based on the pattern of message request/traces. There are two approaches to prefetching, one is to prefetch a set of fixed blocks. The second is to analyze the block which is frequently being requested and is failing to be found in the client cache or global cache. The continuous monitoring of the cache miss of such blocks could help us in prefetching those set of blocks.

Within this paper, we describe the design of our simulation/framework, server, client-server manager and the clients. The thread communications between each component, configuration files and a log file to debug and store results is used in our simulator. We will also lay out the algorithms...
and the evaluation metrics used for the comparative study. Using graphical analysis, we will be able to determine if the proposed prefetching algorithm works better than the stand alone collaborative caching replacement algorithms.

III. RELATED WORK

A. Summary Cache

For each client to know which requested memory block is present whose cache, a dictionary of all cache details is stored. Thus when the block is not found in the requested client’s cache, the dictionary is queried and if it is present in other clients’ cache the request is forwarded to them. The summary cache only updates in intervals (lazy cache), thus there are chances that we have cache fails. Thus depending on the network traffic and the memory required, the summary cache may have some pitfalls.

B. Greedy forward algorithm

In this simple algorithm, when each client work independently without worrying about other clients’ cache content. Thus when a client receives a request it first checks its local cache, if it is not present it forwards the data to the other client. The information about each client and its cache details is stored in a client-server manager as a dictionary. If none of the client is able to serve the request it finally is served by the server. Each time a cache block is replaced or deleted, the global cache dictionary is updated at every run cycle.

C. N-Chance algorithm

N-chance is an extension of Greedy, however in this algorithm clients cooperate with each other to maintain their local cache. In this algorithm, blocks that are not present in any of the cache except one is made to retain for some ‘N’ chances. These singlet blocks when they have to be removed from one client’s local cache is forwarded to another client. The client is chosen randomly to hold on to the singlet. The information about the singlet is stored in the dictionary look up in the manager.

D. Robinhood algorithm

Robinhood algorithm is built to handle the singlets in N-Chance algorithm. Instead of randomly choosing a client to hold on to the singlet, a victim client is chosen using a specified algorithm. A victim block is the one replaced by the singlet. This ensures that the singlets are not continuously forwarded on to different clients.

IV. IMPLEMENTATION DETAILS

A. Simulator Framework Design

The implementation of the three co-operative caching algorithms and the prefetch algorithm is constructed with the same client-server architecture with a client-server manager in between. The simulator is developed in Java and as a multi-threaded environment. The simulator constitutes of four main components i.e., client, client-server manager(csmanger), server and the main memory. The environment and the arbitrary parameters are set using a WarmUp class and the simulator is run using the main class ExperimentSetUp. The WarmUp class reads the configuration file to set input parameters like the number of clients, cache size, the request size etc. The co-operative algorithm is individually set and for each experiment the algorithms are run separately. The framework enables the client to run as separate threads. The client once initiated, starts to send trace blocks to the client-server manager and it gets forwarded to the client also server as per the algorithm set to clients. The client-server manager manages the summary cache which acts as a look up table to forward the client requests. If the requested block is found in another clients cache the request is forwarded to the server or to the server. Once the request is served, the summary cache is updated. The cost to serve each of the requests is summed up and stored in the Requests class. Along with the cost, the local hit rate, global hit rate and server hit rate is also stored. The cost to serve the request in client caches system is lesser than it by served by the server or main memory.

The Request and Trace class are used to store all the information about the trace block being served. The cumulative tick value, hit rates and the path is used to analyze each of the algorithms. The Algorithm abstract class holds the implementation of forwarding the request to the CSManger[3]. The Greedy, nChance and robinhood algorithms class extend the Algorithm base. The logic of each of these algorithms are written in the implementation function. The simulators Cache class contains a list of blocks that represent the cache object of the client. Each block is replaced as per the First-In-First-Out cache replacing algorithm. The list also contains the information of how many times the singlet block has been forwarded. The Client class extends the Java Thread class and has on object of trace class, request class and algorithm class. As per the algorithm set the client class send requests to the CSManger and the results are stored. The CSManger class handles the summary cache list i.e., the details of where a block is located. The class returns a list of client ids which have the requested trace block. If the block is not present in the summary cache it is then forwarded to the server. This class also updates the summary cache once a request has been served. The prefetching algorithm to analyze the block which have been frequently missed or used is run every 15 counts of the CSManger being called. The Constant Enumeration of the simulator hold the constant variables for the algorithm names and the experiment types. The simulators WarmUp class reads the configuration file and sets the respective values of the client size, cache size, server cache size, trace size and cost of serving a request from a client or server and the number[4]. This class is responsible for arbitrarily filling up clients cache, individual trace requests and server cache. This helps in performing the baseline experiments and the comparative study of different caching algorithms and prefetching approaches. The Server class acts as the main storage of all the blocks and if none of the clients are able to serve the requests then the requested trace block is served by the server. The entire
The simulator is run by ExperimentSetUp class. The clients are made to run as individual threads and the values of each of the metrics is then written out to a csv file.

B. Simulation Workflow

1) Client removes a block request from the trace requests.
2) The client first searched for the block in its own cache.
3) If the block is not served by the client’s cache it is then forwarded to the CSManager class.
4) The CSManager checks in the summary cache if the requested block is present in any of the other clients.
5) If the block is not found in step 4, it is forwarded to the server and the summary cache is updated. The request class metrics are updated with the cost to server the request.
6) If the block is found in the step 4, then the CSManager sends the client list which might contain the requested block.
7) The requesting client then forwards the block one-by-one to the clients present in the list.
8) If the block is found in one of the client list then the tick cost is updated and the block has been served.
9) If the block is not found in step 8, then the requesting client forwards it to the server.
10) The block once served is then placed in the requesting client’s cache using the FIFO replacement algorithm and as per the co-operative algorithm the singlets are handled.
11) The prefetching algorithm is run every 15 counts of the CSManager being called.
12) The metrics after a simulation are cumulatively written out to a file.

C. Greedy Co-operative Algorithm Implementation

The Greedy Java class contains the implementation of this algorithm. This class extends the Algorithm abstract class. The implementation function first checks whether the requested block is present in the client’s own cache. If not present the block is encapsulated in the Trace class and sent to the CSManager class. The CSManager handles the request as per the aforementioned logic. Once the request is served, the response is sent back to the Greedy class. The requested block is then stored in the client’s cache. The block to be replaced by performed using the normal FIFO algorithm.

D. nChance Co-operative Algorithm Implementation

The requested block in nChance algorithm is handled in the same way as it was in the Greedy implementation, excluding the part where the block is replaced in the client’s own cache. Once the requested block is found by the CSManager’s assistance, the element to be replaced in the requested client is first checked for a singlet block criterion. The CSManager’s summary cache hashmap is iterated over to find if the block to be replaced is a single cached copy. This is done by checking the size of the client list that is returned. If the client list size is one, then the block to be replaced is a singlet. Subsequently, the ‘n’ value of the block is also verified in the client cache. If the value is greater than 0, then an iterative replace function is called which forwards the singlet to a random peer client. The ‘n’ value of the singlet block is decremented by one. If the singlet’s ‘n’ value is 0, then the block is discarded and the requested block of the client is stored in the client cache.

E. Robinhood Co-operative Algorithm Implementation

The Robinhood algorithm class also handles the singlet block’s ‘n’ values like nChance, however the iterative replace works differently here. The algorithm we have chosen to replace a block works in two folds. First, we choose a block which has been cached multiple times. This is done by iterating over the CSManager’s summary cache hashmap and finding the block whose client list is the maximum. If there is a tie on two blocks, we randomly select one block from which we select a client to forward the block to. In case, all the blocks are singlet, we then choose a client which has space to accommodate the singlet block. This is implemented with the help of the client list. This list is iterated and the cache object size of each client is compared. In this way we select a victim block and victim client to forward the singlet block.

F. Prefetching with Co-operative caching Algorithm Implementation

The prefetching algorithm is run every at definite intervals of CSManager being called. The function of prefetch algorithm logs all the information about the frequency of the trace requests in hashmap called traceMap. Thus, the traceMap contains the block number as key and its frequency as value. The traceMap is sorted by value and the top five frequently requested trace blocks, which are not present in the summary cache, are chosen to prefetch. The blocks chosen are then sent to clients for them place it in their own cache. This approach is to make sure that the latency to get a block is reduced and also the cost/tick counts to get a block is reduced.
V. EVALUATION METRICS

A. Cache Hits and Disk Access

The goals of cooperative caching is to have maximum cache hits and minimum disk accesses possible because disk access is expensive.

B. Block Access Time (Ticks)

A tick is a measure of movement of a block of data in cooperative caching system. The disk access ticks are the most expensive ticks, as disk access is an expensive and slow process.

C. Number of Clients and Cache Size

In our simulation we have designed it to run for various values of client size, cache size and trace size. Varying the cache and the client size we see how the cache hits and disk access varies. The analysis of which will help to prove the hypothesis.

D. Local Hit Rate

The number of times the requested block is found in the client’s own cache. This is compared against the number of clients and cache size. The local hit rate of each of the experiments are set as a benchmark.

E. Global Hit rate

The number of times the requested block is found in the global/summary cache. This helps us to understand how many times the summary cache has failed to provide the right set of client list to the requesting client. If there is a miss in summary cache then the summary cache has either failed to update or the block is actually not present in any of the clients. This metric helps us evaluate the efficiency of the summary cache system.

F. Server Hit rate

The number of times the requested block is found in the server. The server hit rate increases in the worst case scenario.

VI. EXPERIMENTAL SET UP

A. Input Values

The design of the experiments completely depends on the input parameters we provide to the system. The main parameters to send to the system is from the configuration file. The parameters namely number of clients, cache size, server cache size, trace requests size, cost to move a block from client to client, client to CSManager and client to server. The tick costs are assigned in a way that client to client, client to CSManager have lesser weightage than the request being served by the server. These parameters are varied in every experiment to attain the best, average and worst-case scenarios. The number of clients are varied from 200 to 1000, cache size from 0 to 1000 and trace requests from 100 to 1000. The algorithms greedy, nchance, robinhood are studied individually and then along with pre-fetching algorithm. This helps us to learn if pre-fetching technique upheaves the co-operative caching algorithms. The output of each of the experiment is saved as a csv file and using graphical representation the output data is analyzed.

B. Data Collection

As we need to simulate baseline scenarios we populate the trace requests in the following in three ways:

1) Best case scenario - Message requests are populated from the client cache itself. This ensures that the request will definitely be handled by the client’s own cache.

2) Average case scenario - Message requests are populated from the summary/global cache. This ensures that the request will definitely be handled by the client’s own cache or the global cache.

3) Worst case scenario - Message requests are populated from the server cache. Thus, we will know in the real world scenario how the collaborative caching will perform.

VII. GRAPHICAL ANALYSIS

This experiment will provide us information about how on increasing clients size the performance metrics changes. For the first experiment we assigned the following values to the input parameters:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of clients</td>
<td>200 - 1000</td>
</tr>
<tr>
<td>Cache Size</td>
<td>1000</td>
</tr>
<tr>
<td>Number of Message Requests</td>
<td>500</td>
</tr>
<tr>
<td>Cost in Client’s cache</td>
<td>1</td>
</tr>
<tr>
<td>Cost Client to CSManager</td>
<td>2</td>
</tr>
<tr>
<td>Cost CSManager to Server</td>
<td>5</td>
</tr>
</tbody>
</table>

To baseline the linear function of each of the graphs as the client size is continuously increasing, we use the following two formulae:

\[
\text{Average Cost per client} = \frac{(\text{Number of Message found in local cache} \times \text{Local tick cost}) + (\text{Number of Message found in global cache} \times \text{global tick cost}) + (\text{Number of Message found in server cache} \times \text{server tick cost})}{\text{Total Number of clients}}
\]

\[
\text{Average Local, Global/Hit Rate} = \frac{\text{Total Number requests}}{\text{Total Number of clients}}
\]

Fig. 4. Best case - Local Hit Rate vs Number of Clients
Fig. 5. Best case - Global Hit Rate vs Number of Clients

Fig. 6. Best case - Server Hit Rate vs Number of Clients

Fig. 7. Best case - Ticks vs Number of Clients

Fig. 8. Average case - Percentage Local Hit Rate vs Number of Clients

Fig. 9. Average case - Percentage Local Hit Rate vs Number of Clients

Fig. 10. Average case - Global Hit Rate vs Number of Clients

Fig. 11. Average case - Percentage of Global Hit Rate vs Number of Clients

Fig. 12. Average case - Percentage of Server Hit Rate vs Number of Clients
For the first set of experiments we have populated the message requests of each client from the clients cache itself. This ensures that we will get the blocks from the clients own cache. This would be the best scenario sets a baseline of our approach.

In Fig.4 we can see that with prefetching algorithm's baseline for Local Hit rate vs Number Of clients. Thus, this helps us to know that with prefetching algorithm along the co-operative caching works the same and system is not distorted.

In Fig.5 the global hit rate is a straight line at 0 as the clients get the requests from its own cache itself. The clients do not have to access the global or server cache.

In Fig.6 the server hit rate with prefetching is also a straight line at 0 as they clients get the requests from its own cache itself. All the algorithms are in a straight line.

In Fig.9, after prefetch the greedy algorithm looks to work better, as the CSManager is prefetching mostly the singlets which are being discarded by the greedy algorithm. In nChance and Robinhood we see that there is not much improvement with the pre-fetching algorithm.

In Fig.10, after prefetch the greedy algorithm looks to work better, as the CSManager is prefetching mostly the singlets which are being discarded by the greedy algorithm. In nChance and Robinhood we see that there is not much improvement with the pre-fetching algorithm.

In Fig.11, we have the tick values starts around 950, and slowly decreases to around 750 and number of client increases. As number of clients increases size of global cache increases and most of the blocks becomes available in either local or global cache. The cost converges faster in case of pre-fetching for greedy algorithm, nChance and robin remains does not have any effect.

VIII. RESULT

Our hypothesis to find if pre-fetching algorithms works well with collaborative caching algorithms is answered well using the graphical analysis. The pre-fetching algorithm works best with Greedy algorithm as the singlets are not handled in this algorithm. The CSManager being a global cache prefetches those blocks are globally used by all clients. Thus we see that pre-fetching algorithm does not distort the system and works well when paired with Greedy Algorithm.

REFERENCES