Analysis of Collaborative File Caching Algorithms

by

Pearl Rodrigues

A Project Report Submitted
in
Partial Fulfillment of the
Requirements for the Degree of
Master of Science
in
Computer Science

Supervised by

Dr. Hans-Peter Bischof

Department of Computer Science

B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, New York

May 2017
Abstract

Analysis of Collaborative File Caching Algorithms

Pearl Rodrigues

Supervising Professor: Dr. Hans-Peter Bischof

In this paper, we study the behavior of collaborative cache for files. Access from a cache is very quick, however, the size of the cache for a machine is limited. Hence, here we use a collaborative cache, which is a combination of the cache of many machines, creating an illusion of one large cache.

A file is stored as a set of blocks in the memory. Traditionally, when a file is requested, the blocks of that file are requested one at a time. In our approach, we are assuming that we know which blocks belong to a file, and on requesting for one block, we are then requesting all the subsequent blocks. Simulations and results will show the effectiveness of our scheme.
Contents

Abstract ................................................................. ii

1 Introduction ......................................................... 1

2 Related Work ....................................................... 3

3 Cache Algorithms .................................................. 4
   3.1 First In First Out ................................................. 4
   3.2 Last In First Out ................................................. 4
   3.3 Least Recently Used .............................................. 5
   3.4 Most Recently Used .............................................. 5
   3.5 Random ............................................................ 5

4 Assumptions ......................................................... 6

5 Design ............................................................... 7

6 Implementation ...................................................... 9
   6.1 Simulation of Time ............................................... 9
   6.2 Simulation of Trace Files ....................................... 9
   6.3 Properties ....................................................... 10
      6.3.1 Sample Configuration File ................................. 12
   6.4 Work Flow ...................................................... 12
   6.5 Major Classes .................................................. 14

7 Experiments ........................................................ 16
   7.1 Block Sets of Size 1 .............................................. 16
      7.1.1 FIFO ...................................................... 17
      7.1.2 LIFO ...................................................... 18
      7.1.3 LRU ...................................................... 19
List of Figures

1.1 Collaborative Caching ................................................................. 1
5.1 Simulator Design ........................................................................ 7
6.1 Simulator Work Flow ................................................................... 13
7.1 FIFO: Block Sets of Size 1 .......................................................... 17
7.2 LIFO: Block Sets of Size 1 .......................................................... 18
7.3 LRU: Block Sets of Size 1 ............................................................ 19
7.4 MRU: Block Sets of Size 1 ............................................................ 20
7.5 RANDOM: Block Sets of Size 1 .................................................... 21
7.6 Block Set Size 1: Block Processing Time > Block Retrieval Time .... 22
7.7 Block Set Size 5: Block Processing Time > Block Retrieval Time .... 23
7.8 Block Set Size 9: Block Processing Time > Block Retrieval Time .... 24
7.9 Block Set Size 13: Block Processing Time > Block Retrieval Time .... 25
7.10 Block Set Size 275: Block Processing Time > Block Retrieval Time ... 26
7.11 Block Set Size 1: Block Processing Time = Block Retrieval Time .... 27
7.12 Block Set Size 5: Block Processing Time = Block Retrieval Time .... 28
7.13 Block Set Size 9: Block Processing Time = Block Retrieval Time .... 29
7.14 Block Set Size 13: Block Processing Time = Block Retrieval Time .... 30
7.15 Block Set Size 275: Block Processing Time = Block Retrieval Time .... 31
7.16 Block Set Size 1: Block Processing Time < Block Retrieval Time .... 32
7.17 Block Set Size 5: Block Processing Time < Block Retrieval Time .... 33
7.18 Block Set Size 9: Block Processing Time < Block Retrieval Time .... 34
7.19 Block Set Size 13: Block Processing Time < Block Retrieval Time .... 35
7.20 Block Set Size 275: Block Processing Time < Block Retrieval Time .... 36
Chapter 1

Introduction

Cache memories are high speed buffer memories. Data present in the cache can be accessed in ten to twenty-five percent of the time taken to access data in the main memory [5]. A block is a sequence of bits or bytes, present in the memory. Data is generally stored as sets of blocks. By doing so, overhead is reduced and data can be processed faster.

In regular file caching, a request is sent to the client for a particular block, and once it is processed, a new request is made out for the next block. Typical files, having a size raging from a few kilobytes to a couple hundred megabytes, consist of 5 to 20 blocks. However, bigger files, such as movie files, which can range from a few hundred megabytes to a couple of gigabytes in size, can constitute over 400 blocks.

![Collaborative Caching](image)

Figure 1.1: Collaborative Caching
Given a group of network caches, we need to come up with a way to utilize all the available cache efficiently, in order to maximize performance. In this scenario, maximizing performance would mean more hits than misses. A hit occurs when the file block that a client is looking for is present within it’s cache. If the block is not in the client’s cache, and if it needs to fetch that block from an external source, that counts as a miss. Collaborative caching is an approach of distributed caching. Here, several machines are combined together to give an illusion of one large machine [1]. The reason for using collaborative caching is simple- we know that accessing data from the cache is extremely fast, but we also know that we only have a limited sized cache on one machine. Hence, by combining several machines, and hence their caches, we are simulating one large machine with a very large cache. Also, by increasing the size of the cache, the system is bound to have more hits than misses, as there is more room for more file blocks to be stored within the cache.
Chapter 2

Related Work

As mentioned earlier, the traditional approach of caching files goes as follows- a client requests for a particular block of a file, once it receives and processes it, it then makes a request for the next block, and so on. The clear downside here is that the client has to wait for every block to be received after it makes a request for it. Instead, our idea is to enable the client to fetch all the blocks of a file while simultaneously processing them.

When a client requests for one block of a file, while it is processing the first block, the following blocks are also requested simultaneously. Then, depending on how fast (or slow) the processing of a block is, the next block will either be ready to be processed in the buffer, or the client will have to wait some time for it to be added onto the buffer.

The main idea here is that, whether the client has immediate access to the next block or not, by letting it fetch the remaining blocks, in theory, we’ll be able to reduce the amount of time it has to wait in order to process the entire file.
Chapter 3

Cache Algorithms

Cache replacement algorithms are algorithms used to determine what needs to be removed from the cache in order to make space for something that’s being requested. Since the cache is of a limited size, there will be several swaps happening over time in order to keep the cache updated. We’ll start with describing the different cache replacement algorithms that we have used in our simulations. These are just a few of the popular approaches, and there are many more strategies available for the replacement of cache.

3.1 First In First Out

First In First Out (also known as FIFO) is one of the most commonly used cache replacement strategies. This algorithm keeps track of the arrival time of each block. When there is not enough space to accommodate a new block into the cache, blocks currently in the cache are removed in the order that they arrived, until there is enough space for the new block to be placed into the cache [3]. The downside of this algorithm is that if blocks were being requested in the same sequence after a certain number of iterations, we’d end up getting bad performance with the FIFO strategy.

3.2 Last In First Out

Last In First Out (also known as LIFO) works exactly opposite to the FIFO strategy. While this algorithm also keeps track of the sequence of arrival of the blocks, when a block needs to be removed, the one that had entered the cache last is the one that is replaced with the
newly arriving block [4].

3.3 Least Recently Used

Least Recently Used (or LRU) is a cache replacement strategy that keeps a track of the how often a particular block is accessed in the cache. When the time comes to remove a block in order to make room for a new block, the one that was used the least, i.e. had the lowest frequency, is swapped with the new block [3]. A scenario where this strategy provides poor performance is when there is a repetitive sequence of access, i.e. as soon as a block is removed and swapped with another, the one that was removed is requested again.

3.4 Most Recently Used

Most Recently Used (or MRU) is a cache replacement strategy that does the opposite of LRU. The block that is removed using this strategy is the block that was accessed the most recently when in the cache [2]. This strategy fails when we have every alternate request for the same block, and the others filled with new blocks. In this case, it would always remove the one last accessed, and make room for the new block, and then when the old block is requested again, the new block is swapped out, and this continues with a series of misses and swaps for every request.

3.5 Random

This algorithm, as the name suggests, randomly removes blocks from the cache until there is enough free space for the new block to be placed in it [6]. There is no predetermined strategy, or any extra information stored with the blocks in the cache. There is no fixed scenario where this method soars or fails, because it is purely random, and hence the performance is also completely random and cannot be estimated in advance.
Chapter 4

Assumptions

In a traditional file caching approach, one block of a file is requested at a time. For every request, the client needs to locate the file source, and retrieve the required block. We are making two assumptions for our experiments-

1. It is known which blocks belong to a single file.

2. Once one block of a file is requested, all the subsequent blocks of that file will then be requested.

3. No failures occur throughout the simulations.

Firstly, since the focus of our experiments is on the performance of the cache, knowing the blocks of a file help us eliminate the extraneous time it would have taken to locate every block. Moreover, including this additional time would have skewed our results.

Secondly, it is evident that since a file is split up into several data blocks, fetching only a single block would be futile for the client. It is also apparent that in order to process a file, the client would need all the blocks of the file. Consequently, making a request for every succeeding block following the first one is justified.

Finally, since this simulation is focusing on the cache memory and the time taken for requests to be fulfilled, eliminating handling failure caused due to problems like network loss reduces the system overhead.

In our experiments, we’ve simulated with three main scenarios- having the block processing time either greater than, equal to, or less than the block retrieval time. All the results are discussed and explained in the following sections.
Chapter 5

Design

A common framework has been developed in order to run all of the simulations. The framework provides flexibility with respect several entities such as total number of clients, client cache size, server cache size, block retrieval time, block processing time and cache algorithm used. There’s mainly three sets of actors in the framework- a Master, a set of Clients, and a Server. The amount of time taken for every process of the simulator is measured using a Tick, which is defined within the framework.

Once the simulation begins, every active client generates independent requests for different blocks of different files in the system. There are four possible places that the block can be retrieved from- the client’s own cache, another client’s cache, the server’s cache or
the server’s disk. The requests are carried out in the order mentioned, with retrieving the block from the client’s own cache and taking the least amount of time to retrieving it from the server disk while taking the maximum amount of time. Once the first block of a file is requested, the subsequent requests will be for the other blocks of that file. At the end of that set of requests, the entire file would have been processed by the client.

The Master carries out the main simulation and handles the clients and server within itself. At each point of time, there is a fixed number of clients that will be active, and once they’ve all completed a certain number of requests, they are deactivated and the total number of active clients is increased. These clients are then activated and the entire process is run again. For every iteration of the specified number of clients, the average time, processing time and retrieval time is recorded for analysis at a later point of time.

In order to generate requests, trace files have been generated. These files were generated using a machine’s file system, to determine the variation in the number of blocks present in every file. The trace files contain several sets, made up of block numbers. Hence, a set having five block numbers indicates that that block has a set size of five. The block set sizes ranged from 1 to 453. These values reflect the numbers observed on the machine.
Chapter 6

Implementation

6.1 Simulation of Time

Since we are simulating an asynchronous system, we’ve come up with a way to record the time taken for retrieval and processing of the blocks. We’ve created a Tick class, which updates its counter by one at a given interval of time. This interval is provided as a property within the simulation configuration. For all the experiments recorded and discussed in this paper, this interval was set to 10ms.

6.2 Simulation of Trace Files

To get an idea of general file systems and the sizes of various files, we simply recorded the number of files and the respective number of blocks each file was made up of in one of our machines. Using that information, we generated random block identification numbers to create trace files. For instance, if our recording showed that there were 3 files comprising of 5 blocks each, our trace file would be generated to have 3 unique sets of 5 block IDs, that would be used by the clients to generate the requests. We’ve experimented with several trace files, some of which had a mix of different sized block sets, while some just had a constant block set size.
6.3 Properties

There are several properties that can be changed for different simulation environments. Following is a list of the properties-

1. Client end ID
   This property is used to indicate the maximum number of clients that can be running in the simulation before it ends.

2. Increase client size
   The simulation begins with 1 client. After every iteration, the number of active clients is increased by this number, and the process is repeated. The simulation ends when the active number of clients exceeds the client end ID property.

3. Client cache size
   This field is used to indicate the maximum number of blocks the client would be able to hold in its cache at a given point in time.

4. Server cache size
   This field is used to indicate the maximum number of blocks the server’s cache would be able to hold at a given point in time.

5. Cache algorithm
   This property specifies the cache replacement algorithm to be used for the simulation. The allowed values for this property are FIFO, LIFO, LRU, MRU and RANDOM.

6. Tick periodic increase
   The ticker value from the Tick class runs continually in the background during the simulation. This property specifies the number of milliseconds the ticker has to pause before it increments its value.

7. Tick increase own cache
   For every block request, when the client checks for the presence of the block within
it’s own cache, the tick value will be increased by this number.

8. Tick increase other client’s cache
   For every block request, when the client checks for the presence of the block within other client’s caches, the tick value will be increased by this number.

9. Tick increase server’s cache
   For every block request, when the client checks for the presence of the block within the server’s cache, the tick value will be increased by this number.

10. Tick increase server disk
    For every block request, when the client checks for the presence of the block within the server disk, the tick value will be increased by this number.

11. Tick increase processing time
    This property is used to indicate the processing time of a single block. The tick value is increased by this amount every time a block is processed.

12. Tick increase waiting time
    If a block is not present in the client’s buffer so that it can be processed, the processing thread will wait for this number of milliseconds before it checks for the same block again. The tick value is increased by this number every time the thread waits.

13. Tick increase retrieval time
    This property is used to indicate the retrieval time of a single block. The tick value is increased by this amount every time a block is retrieved.

14. Trace file
    This property contains the path to the trace file that will be used by all the clients to generate block requests.
6.3.1 Sample Configuration File

client_end_id=200
increase_client_size_by=3
client_cache_size=500
server_cache_size=500
cache_algo=FIFO
tick_periodic_increase=50
tick_increase_owncache=1
tick_increase_otherclientcache=1000
tick_increase_servercache=5000
tick_increase_serverdisk=10000
tick_increase_processing_time=500
tick_increase_waiting_for_block=500
tick_increase_block_retrieve=2000
trace_file=traceFile.txt

6.4 Work Flow

We’ll explain how the simulation works, one step at a time, in this section. First, the configuration file is read and all the constants are set up. This includes the total number of clients, the interval after which the tick counter will update, the source trace file, the block processing time, and so on. The Server cache is filled up with random sets of blocks from the trace file. Once that’s done, the Master creates the current number of active client threads. This number starts with 1, and goes on until the maximum number of clients set.

Each client operates independently. It generates a request for a block in every iteration. Once a request is made, these are the steps it goes through to retrieve the block-

1. The client checks if the block is present in its own cache. If it finds the block, the retrieval process is complete.
2. If the block was not present in the client’s own cache, it then checks all the other client’s caches for the block at an additional cost of time. If it still does not find the block, it moves on to the next step.

3. Having failed to find the block within the set of clients, the client informs the master that it is looking for that particular block.

4. The master forwards the request to the server, indicating which client is looking for a block, and the block number that’s being requested.

5. The server checks its own cache for the block. If the block is found in the server cache, it is returned to the client.

6. If the item was not found in any of these places, the server finally fetches the block from the disk. This access is the most expensive of all, and will add to the retrieval time of the block by a great amount.
6.5 Major Classes

1. Master: This is the main class that sets the environment up and is responsible for managing the clients. All requests to the server go via the master.

2. Client: Every client is an instance of this class. Each client has its independent thread, that generates block requests and finds the blocks that it needs, either in its own cache, in another client’s cache, or in the server.

3. Server: This is the main server class. It has its own cache, whose size is defined in the configuration. It also has a disk. Any requested block that isn’t present in any of the caches will definitely be found on the server disk.

4. Cache Replacement Algorithm: This is an interface, and there are several classes, namely, FIFO, LIFO, LRU, MRU and RANDOM that implement it. It contains all the basic cache replacement methods, such as add, remove and swap.

5. Simulator Logger: This class is responsible for printing error/information messages in the output files.

6. ReadTraceFile: This class takes care of parsing through the trace file and recording what blocks are present, so that the clients can request for blocks from that set.

7. Tick: This class keeps track of the time. For every few milliseconds, it updates its counter by 1. The amount of time it waits before updating its counter value is defined in the configuration file.

8. Constants: This class is responsible for setting all the constants defined in the configuration file. The Master sets the constants before starting any clients.

9. RetrieveBlock: Every client has a thread of this continually working. As it locates blocks in the network, it adds them to a buffer, such that the ProcessBlock thread will then be able to access and process it.
10. ProcessBlock: Just like RetrieveBlock, every client has a thread of ProcessBlock as well. This thread continually runs in the background, and as the RetrieveBlock thread places new blocks into the buffer, this thread processes them. The amount of time it takes for a single block to be processed is set from the configuration file. If there is no block in the buffer, the thread waits until something is added into it.
Chapter 7

Experiments

In the following sections, several experiments and strategies have been discussed. The block set sizes used in these experiments are 1, 5, 9, 13 and 275. We’ve used block set size 1 because that’s the lowest number that we have, and that is what we can use to lay the groundwork for our experiments. Block set size 275 was chosen because the client and server cache sizes were fixed at 500, and by having block sets of size 275, we were only allowing one set to remain in every cache at a time. Theoretically, this case had to have very poor performance. Block sets 5, 9 and 13 were used because they make up the most common sizes of regular files, such as documents and text files.

7.1 Block Sets of Size 1

Initial experiments were conducted with a fixed block set size of 1, i.e. it is assumed that every file present in the system has only one block. Hence, the trace file only contained block sets with a single block each. The experiments were conducted for up to 200 clients for each replacement algorithm. Following is the set of properties for this set of simulations-

1. Client end ID: 200
2. Increase client size by: 3
3. Client cache size: 500
4. Server cache size: 500
It is assumed that it does not take any extra time for the blocks to be processed in these sets of experiments. The only property that is different for each of these simulations is the cache replacement algorithm used. Following are the observed results-

### 7.1.1 FIFO

In this experiment, we can see that there seems to be a certain amount of fluctuation when there are about ten to twenty five clients, but this can be explained with the purely random nature of the requests. At 10 and 15, there probably were more requests for blocks that were not in the client caches, hence the time taken did not improve when compared to the previous iteration. However, we can clearly observe that starting from 50, the performance stabilizes, and there is little or no difference in the system beyond that point. The stabilized time lies between fetching all the blocks from either the client’s own cache, or from another client’s cache. This is close to being the ideal case, where the client would have every block in it’s own cache. However, considering the fact that we’ve used a distributed collaborative system, this is the ideal case for our network.

![Figure 7.1: FIFO: Block Sets of Size 1](image-url)
7.1.2 LIFO

For the LIFO experiment, we can see some spikes in the graphs at around 30 and 50 clients. Again, as stated earlier, it might simply be because the block requests generated at those times were for blocks in the server cache or disk, and fetching blocks from the server took a great deal of time.
Apart from the small spike at 30 clients, the rest of the graph seems to have a constant decline, and it stabilizes at around 50 clients mark, just like all the other algorithms. Also, the stabilized time lies between retrieving all the blocks from within the client itself, or any other client, but not having to contact the server.
There are several fluctuations noticed in this result set. At around 10 clients, instead of the expected decline in time, there is a slight increase. This can be attributed to the random requests, where the blocks being requested being on the server cache and disk, and the client having to spend extra time fetching each one of the blocks. As all the previous algorithms, the performance stabilizes when about 50 clients are running simultaneously.
7.1.5 RANDOM

Finally we have the results of the random algorithm. As already seen before, there is a spike at around the 25 clients mark, beyond which the decline seems natural, following which the time stabilizes to the ideal time.

7.1.6 Results

As it can be observed from all the five simulations that the amount of time taken to fetch a block stabilizes over time. After a having about 50 clients, the performance remains almost constant, irrespective of the cache replacement algorithm used. Also, the time that the system stabilizes to is the ideal time, with every retrieval being fulfilled by a client, and not the server. Therefore, it would be safe to assume that no cache replacement algorithm seems to work better than another in this case, and even though there are minor variations in their performance, the overall time is almost the same.

Following this observation, for all the experiments to follow, we’ll only be using the FIFO replacement strategy.
7.2 Block Processing Time > Block Retrieval Time

This section is going to consider several scenarios, with various block set sizes. The common property among all the results in this section is that the block processing time is greater than the block retrieval time. This is an ideal case, because we’d rather have the blocks ready to be processed than to have to wait for them to enter into the client’s buffer.

7.2.1 Block Set Size 1

![Graph showing processing and retrieval times]

Figure 7.6: Block Set Size 1: Block Processing Time > Block Retrieval Time

This is a case where block sets of size 1 have been used in the trace files. The wait and retrieval times set in the configuration are different from the ones used earlier, which is why the system stabilizes much earlier than it did in the previous cases. Clearly, since the processing time is higher, the plot of processing time is way higher than the retrieval time. The total time is the total processing and retrieval time together. As observed in the previous set of results, once the time stabilizes to the ideal rate, the average performance remains the same, irrespective of the number of clients in the network.
7.2.2 Block Set Size 5

Just like the previous case, the plot for processing time is higher than retrieval time because of the set parameters. Compared to the block set size 1 experiment, it takes a longer time for the system to stabilize, because for every set request, there is five times more requests being sent to the server when the block isn’t found on any client, when compared to the trace file having just block sets of size 1.
7.2.3 Block Set Size 9

As observed in the earlier results, this is again an expected case, where the trace file used has block sets of size 9. It starts off with having a high processing, retrieval and hence total time, but eventually it stabilizes to an ideal point where most of the requests get fulfilled by the clients themselves.
7.2.4 Block Set Size 13

Figure 7.9: Block Set Size 13: Block Processing Time $>$ Block Retrieval Time

Having a block set size of 13 is also quite common in regular files. Compared to all the smaller block sets seen so far, this takes the most time to stabilize. Beyond having about 80 clients in the network, the overall time to fulfill requests ends up being close to constant.
7.2.5 Block Set Size 275

This experiment, by far, shows the worst performance in the network. The time taken does not even stabilize on time. This is because the client cache size is limited to 500 blocks, but here when each set of size 275, every client is only able to retain one set in its cache at a time. As a result, almost every request gets fulfilled by the server, and even having 200 clients does not improve the performance. As it can be seen, on an average, the time to fulfill a request lies between fetching blocks from the server cache, or the server disk, neither of which is the ideal case.

7.2.6 Observations

Since we have set the block processing time to be higher than the block retrieval time, we can observe the plot of retrieval time is always below the processing time.

The interesting observation is that for block sets of sizes 1, 5, 9 and 13, the performance seems to stabilize after a having a certain number of clients in the system. And beyond that point, we do not see any improvement in performance. Each of these simulations stabilized
at different times, with the smaller block set sized experiments stabilizing with just about 10 clients, while the larger sets needed about 80. Hence, the number of clients required for the performance to stabilize is directly proportional to the block set size being handled.

The set with 275 blocks is clearly an exception, and performs the worst. This is because the client cache size is limited to 500, and hence at a time each client is only able to hold one set in its cache. Owing to this disadvantage, all the requests need to be fulfilled by the server, hence leading up to extremely poor performance.

7.3 Block Processing Time = Block Retrieval Time

The common property among all the results in this section is that the block retrieval time and the block processing time are both the same.

7.3.1 Block Set Size 1

![Figure 7.11: Block Set Size 1: Block Processing Time = Block Retrieval Time](image)

Owing to the configuration settings, the processing and retrieval plots overlap, since both block retrieval and block processing time are set to the same value. As established in the
previous set of experiments, the performance stabilizes rather quickly when there is only one block in every block set of the network.

### 7.3.2 Block Set Size 5

![Graph showing Block Processing Time vs. Block Retrieval Time]

**Figure 7.12: Block Set Size 5: Block Processing Time = Block Retrieval Time**

The retrieval and processing plots do not exactly overlap in this experiment, but that can be explained by adding the amount of time the processing block has to wait in case the block to be processed is not in the buffer. This situation could arise when there might have been a slight delay in the start of the retrieval thread, because of which the processing thread then is forced to wait for the block to arrive in the buffer.
7.3.3 Block Set Size 9

![Block Set Size 9: Block Processing Time = Block Retrieval Time](Image)

Figure 7.13: Block Set Size 9: Block Processing Time = Block Retrieval Time

For block sets of size 9, the processing and retrieval times almost overlap. When there are about 60 clients in the network, as seen earlier, the time starts to stabilize, reaching its ideal performance time.
7.3.4 Block Set Size 13

This set, as already seen, takes the longest to reach its stable state. Here, it takes having about 80 clients running simultaneously to get to that state. However, once it reaches there, the time remains within the boundaries of access from clients caches.

Figure 7.14: Block Set Size 13: Block Processing Time = Block Retrieval Time
7.3.5 Block Set Size 275

This final experiment, as expected, has the worst performance in the entire set. As mentioned earlier, since the cache size is not too much, at a time only one block set has the space to be on the cache, because of which all requests end up getting forwarded to the server, leading to poor performance.

7.3.6 Observations

In this set of experiments, we’ve set the block processing time and the block retrieval time to be the same. Hence we notice that the retrieval and processing plots overlap in almost all the cases, except for block set of size 1.

This is because for the block sets having just a single block, even though the retrieval and processing time is the same, the time penalty in case of a miss is high. This adds up for the processing time because the processing thread then has its ticker incremented. For larger block sets, this number becomes negligible because there are several blocks to
process, but since we only have a single block in the first case, the miss time penalty ends up having a higher priority.

### 7.4 Block Processing Time $< \text{Block Retrieval Time}$

The common property among all the results in this section is that the block retrieval time is greater than the block processing time.

#### 7.4.1 Block Set Size 1

![Figure 7.16: Block Set Size 1: Block Processing Time $< \text{Block Retrieval Time}$](image)

In this first experiment, with a block set size of 1, the results are just as expected. The plot of the retrieval time is above processing time, and this is because the configuration is set in that manner. As already seen in the past two sets of experiments, this system stabilizes very quickly, and once it reaches that ideal state, the performance neither improves nor goes down.
7.4.2 Block Set Size 5

Contradictory to the previous experiment, for block sets of size 5, the plots for retrieval time and processing time overlap, even though we’ve set the retrieval time to be higher. This is because it is most likely that both the threads started at the same time, and since the processing time cannot proceed without having a block to process, it waits for the entire retrieval time to begin its work. And since the processing time itself is much lesser than the retrieval time, the total processing time adds up to be almost the same as retrieval time. As seen earlier, the system stabilizes over time, taking slightly longer when compared to the previous experiment with smaller block sets.
7.4.3 Block Set Size 9

Figure 7.18: Block Set Size 9: Block Processing Time < Block Retrieval Time

The results for block sets of size 9 are almost exactly like the last experiment. As already mentioned, the processing thread waits for the amount of time it takes for retrieval to be completed, as a result of which both the plots overlap. This system, as for the previous sets of experiments, stabilizes on having about 60 clients.
7.4.4 Block Set Size 13

There are a few fluctuations in the plots for this experiment, but that could be attributed to the random generation of requests, because of which there might be cases where blocks that are on the server were requested more than the others. This, in turn, ends up creating a spike in the time taken to fulfill that set request. As noticed earlier, the processing and retrieval plots align almost perfectly, and the system stabilizes after having around 80 clients.
7.4.5 Block Set Size 275

Figure 7.20: Block Set Size 275: Block Processing Time < Block Retrieval Time

In this final experiment, again we can see the pattern of having both the processing and retrieval plots on top of one another. For this larger block set size, the performance is extremely bad, and the system does not stabilize even with 200 clients in the network. This is because of the cache size being limited to 500, hence restricting every client to not hold more than one block set, which comprises of 275 blocks, at a time.

7.4.6 Observations

We’ve set the block retrieval time to be higher than the processing time. However, in almost all of our graphs, the processing time and retrieval time plots coincide. This is because even though a block can be processed fast, the processing thread needs to wait for that block to become available in the buffer. Hence, the total processing time ends up being roughly the same as the retrieval time as the time it takes for just processing is negligible.

Also, as observed earlier, block sets of sizes 1, 5, 9 and 13 behave in the same manner, and the set with 275 blocks performs the worst.
Chapter 8

Conclusions

There are a number of things that can be concluded from all our simulations and experiments.

1. When it comes to analyzing the performance of collaborative cache, if the block requests are truly random, the cache replacement algorithm used is irrelevant. All the algorithms end up performing almost the same, with a few variations every now and then.

2. The performance of a distributed collaborative cache system will stabilize after having a certain number of clients. Increasing the number of clients beyond that point will not improve performance.

3. The amount of time it takes for the performance to stabilize is directly proportional to the size of the block set.
Chapter 9

Future Work

1. In the current simulation environment, trace files are generated ahead of time, and they are used to generate every block request. This could be extended to dynamically generating requests using the machine’s file system.

2. Rather than sending empty blocks, the actual file blocks from the machine could be sent. The new patterns of block requests being fulfilled could then be compared against the current simulation.

3. Additional or hybrid cache replacement algorithms could be incorporated, and one could observe if any of the newer algorithms contribute to the performance of the system.
Bibliography


