Analysis of N-Block Caching Strategies

Rajashree Krishnadevamaiya∗

Department of Computer Science, Rochester Institute of Technology

ABSTRACT

In this paper, we study the behavior of distributed cache for N-block replacement in cache replacement strategies. Distributed cache is an extension of traditional concept of cache. A distributed cache spans multiple servers, which can grow in size and transactional capacity, providing better consistency and reliability compared to in-memory caches. Traditional cache replacement strategies replace one block at a time, we study the effect of replacing N blocks at a time. In this new approach to distributed cache management, we will evaluate cache replacement algorithms for hit and miss rates, memory access time, and disk access rate. In this paper, we describe the techniques used to implement cache replacement strategies for N-block replacements. Simulation and visualizations are used to show the effectiveness of the proposed scheme.

1. INTRODUCTION

Cache memory enables faster access of the data/instructions that are frequently accessed by the software in a computer. The idea of cache memory is to hold the data that is currently being used temporarily in a high speed memory that improves the overall performance of a system. The time required to access a cache memory is 20-25% lesser than accessing the main memory [5]. In simple caching, the requesting node performs the caching and then the cached copy of the data is used in the subsequent incoming request. The requesting node does all the operations including fetching the data from disk/main memory. In this paper, we implement a distributed cache. Distributed caching allows for sharing of cached data between clients or devices. It provides better consistency and reliability compared to in-memory caches. It allows sharing of cached data between the clients, which in turn leads to lesser access to disk/main memory.

In this paper we propose a N-Block replacement scheme which is an extended concept of the traditional single block replacement scheme in simple cache. This is implemented using a simulation technique and evaluated or the standard performance metrics for a cache. We have implemented four different algorithms in our simulation framework to study the behavior of single block and N-block replacements respectively. We further propose our own modified version of the NChance algorithm which is then compared with the previous work. In this paper, we describe the design of our simulation framework, architecture and the components of the simulator in detail. The clients, Content Manager, thread communication between the clients, a logger used for debugging, various scripts, server, and a main memory are used in our simulator design. We describe the basic workflow of simulation and experimental setup in the further sections. We also show the results of our simulation in the form of graphical analysis to describe the behavior of each simulation run.

2. RELATED WORK

2.1 First In First Out

First In First Out (FIFO) replacement algorithm is the simplest and has the least overhead. This algorithm is also known as the Round Robin replacement. The FIFO algorithm keeps track of when a data block was added to the memory. The data block that is the oldest is chosen for replacement, the block that has been in the memory for the longest time is replaced by the algorithm.

2.2 Least Recently Used

Least Recently Used (LRU) algorithm discards the least recently used first. This algorithm keeps track of what was used recently. This is typically implemented using an array or list data structure to keep a counter of what was used when and costs an execution overhead to maintain this data structure. Every time the cache is updated, this data structure also has to be updated. There are several other variations of this algorithm that can be implemented to improve the run time efficiency.

2.3 Least Frequently Used

Least Frequently Used (LFU) replacement algorithm keeps track of number of times a data block is referenced in the memory. When the cache is full, the algorithm chooses the data block that has the lowest reference frequency. LFU is efficient when used in static environments where the frequency of elements do not change too much over a period of time. For a cache that has too many changes, the cost of implementation will also increase. The other disadvantages are the implementation itself is complex as compared to other replacement strategies, and there is an issue of similar values. When the frequency of more than one element is the same, we need some sort of tie breaker. [4]
3. IMPLEMENTATION DETAILS

3.1 Simulator Framework

3.1.1 Design

Our implementation of all five algorithms is based on the same underlying architecture. The simulator is designed using Java. The simulator consists of four different components as shown in Figure 1. A client object, a content manager, a server, and the memory/disk. The client, server, and the content manager are Java classes. The simulation is driven by a main class SimulatorDriver that takes input parameters and starts the simulation. Each client is a Java thread and has its own cache object, tick count object for manipulating tick counts. Each client has the capability to interact with other clients, the content manager, and the server. The content manager is like a look up table that provides information about each client’s cache. This is updated every time any client’s cache gets changed. The content manager also consists of few global data structures that are used in specific algorithms. The server component acts a second level cache in our architecture, if the block requested is not present in client’s local cache or any of the other clients, the request is forwarded to the server cache. Fetching data from the server cache is less expensive than fetching data from the disk. The disk is the main memory where all the data resides, if the requested block is not found in any of the caches, the data is fetched from the disk. The simulator accepts the configuration file with input parameters as a command line argument. The input configuration file is a java properties file that sets input parameters for each simulation. Our main input parameters are local cache size, number of clients, server cache size, algorithms, and the cost of each operation which is represented by tick counts.

Implementation of each Java class is described below:

The SimulatorCache class contains a list that acts as the cache object for each client. This class also contains a secondary list which holds the unique pattern identifiers. These pattern identifiers differentiate between the single data block and set of data blocks.

The SimulatorClient class defines the behavior of each client and handles all the cache operations. Each client is a Java thread, so this class extends the Java Thread class. Each client has its own SimulatorCache object, and a SimulatorTickCounts object to measure the memory access time. All the local objects of this class are initialized at the beginning of the simulation and with creation of each client thread.

The SimulatorConstants class creates constant variables for the initialization of input parameters for each simulation. These parameters are read from an input configuration file and set by the SimulatorDriver for each simulation.

The SimulatorContentManager class is our lookup table which contains information about each of the clients and the contents of their cache. This class also contains global data structures used in NChance algorithm’s implementation. This class contains methods for the data lookup and updating the lookup table every time any client’s cache is updated. The lookup table is a Java map that has client number as the key and the client’s cache as the value.

The SimulatorDisk class acts as the main memory, it has a Java thread and has its own cache object, tick count object to measure the memory access time. Every client has its own tick count object.

The entire simulation is driven by the SimulatorDriver class which initializes all input parameters, creation of clients and the start of each client operation. This is the single entry point in our simulator.

3.1.2 Simulation Workflow:

Figure 2 shows processing of a single request in the simulation.

Client 1 receives an incoming data request:-

1. The client searches its own cache.
2. If the requested is not present in the client’s local cache, the client sends a request to the content man-

![Figure 1: Simulator Architecture](image-url)
3. The content manager responds with a true if the data is present, or false if the data is not present.

4. If the data is present in any other client (Client-N), Client-1 sends a request to Client-N to fetch the data.

5. Client-N returns the data to Client-1.

6. If the content manager returned false in step 3, then Client-1 sends a request to the server for the data.

7. The server searches its own cache, if the data is present, it returns the data to Client-1.

8. If the data is not present in the server, the server sends a request to the disk for the data.

9. The disk returns the data to the server.

10. The server sends the data back to Client-1.

3.2 N-Block Implementation

The simulator was designed to handle N-block replacements. The SimulatorCache class contains a secondary list which holds the unique pattern identifiers. These pattern identifiers differentiate between the single data block and set of data blocks. There are certain design trade offs with this implementation. Our implementation of each algorithm heavily makes use of data structures. There is an overhead of maintaining these data structures and on the run time of the algorithm.

The NChance algorithm takes the longest among all the algorithms since it uses an additional data structure to keep track of recirculation count. This increases the run time of the algorithm since this data structure has to be updated for every data request. The use of lookup table and the recirculation array together increases the run time by a large factor.

However, our design does not take time complexity into consideration, we implement a straight forward approach of N-Block implementation using the unique pattern methodology described in section 3.1.1.

3.3 NChance-K Forwarding

The NChance-K forwarding is the implementation of modified version of N-Chance forwarding algorithm. It keeps track of the frequency of access of each incoming data request and calculates the rate of occurrence of that data block at that point in time. We use this frequency to assign the value of n. In the N-Chance forwarding algorithm, the optimal value of N is known to be 3. But in our implementation we assign the value of N based on the rate of occurrence of each element. The N value we assign are either 3, 4, or 5 based on the frequency of the incoming data block. If the incoming data block has a higher frequency of usage, we assign the largest N, and if it has lower frequency, we assign to the smaller N. This is based on the idea that if the frequency is higher, then the block is treated as a singlet for a longer period of time. We give the block more chances to survive. The performance was compared with the NChance forwarding algorithm, the results are described in the later section.

3.4 Evaluation Metrics

3.4.1 Hit Rate

Hit rate of a cache is defined as the percentage of memory accesses that are satisfied by the cache. In our design we have two levels of cache. The client’s local cache is smaller in size but provides a faster memory access, and a server cache which is larger but has a higher memory access time. A high performing cache will have a higher hit rate.

1. Local Hit Rate Local hit rate is the percentage of memory accesses that are satisfied by the client’s local cache.

2. Neighbor Hit Rate Neighbor hit rate is the percentage of memory accesses that are satisfied by the cache of other clients in the system.

3.4.2 Miss Rate

Miss rate of a cache is the percentage of memory accesses that are not satisfied by the cache. In our design, if the requested data is not found in client’s local cache or any of the neighbor’s cache, it is considered as a cache miss.

3.4.3 Memory Access Time

Memory Access Time is defined as the time taken to fetch the requested data. In our simulation we measure the access time using the concept of "tick counts". Since our implementation is a java simulation, measuring clock time was not a feasible option. Tick counts are defined based on the standard real world design. Memory access time will be low if the data being requested in the local cache, and higher when the data has to be fetched from the disk. Fetching a data block from any given location involves cost of searching and communication to reach the particular location. In an actual implementation of cache the communication time may vary depending on the hardware and processing capabilities of the machine.
3.4.4 Disk Access Rate
Disk access rate is defined as the percentage of data requests that were satisfied by fetching the data from the disk. A high performing cache should have minimum number of disk accesses.

4. EXPERIMENTAL SETUP

4.1 Data Generation
In our experiments, we use data that is generated based on a standard cache input trace taken from [3]. We observed that the data in this cache input trace displayed a pattern of set of blocks that always appeared together. We use this as the basis to generate input data for our experiments. We designed a data generation script using Matlab which generates data in a random fashion with a set of blocks that appear together always at random locations in the input trace file. These input trace files were generated three separate times for different concentration of set of blocks in each. We varied this occurrence of blocks as 5%, 10%, 15% of the total data requests. For example, if there are 100 data requests, at least 5 of them will belong to some set of blocks that always appear together. This was also designed based on the standard cache input trace. We also assign a unique pattern number to each data block which helps us identify whether or not it belongs to a set of blocks or not. This pattern number is used as unique identifier by the replacement algorithms for N-block replacements. if a certain pattern is a single data request of its own, this is treated as 1-block.

4.2 Input Parameters
Our simulation takes 6 main input parameters that are defined in a java properties file. These inputs are read by the simulator driver for each simulation. The six parameters that we use are local cache size, server cache size, number of clients, server replacement algorithm, client replacement algorithm, and the input trace file name.

One of these parameters is varied in every simulation and the output results are saved a csv file. In our experiments we vary cache sizes from 32 to 2048 as powers of 2, number of clients from 50 to 500 in the intervals of 50, and the algorithms are FIFO, LRU, LFU, and NChance forwarding.

5. RESULTS AND DISCUSSIONS
Evaluation N-block caching strategies was done using the distributed cache simulator. These results were the output from the simulations in the form of CSV files. These files are parsed and relevant visualizations are generated. The parameters that are varied in the experiments are local cache size, number of clients, server cache size, and the algorithms used. We observed that when number of clients and local cache size are varied, the results produced can be represented as visualization, whereas varying server cache size did not show different results for single block vs. N-block. The results described below are a subset of results from over 400 simulations. We decided to show the results for the input data that comprised 10% of the entire data was the set of blocks. The results are organized based on the each metric and algorithms in the sub sections.

5.1 Memory Access Time
The goal of improving cache performance is to reduce the memory access time. The time taken to fetch a data block is the least when the data block is present in the cache, and the time taken is the highest when data block has to be fetched from the disk. The results are visualized for varied cache size and number of clients. In each graph x-axis represents the local cache size from 32 - 2048 in log scale, number of clients from 50-500, and the y-axis represents the access time calculated using tick counts.

5.1.1 First In First Out
This section shows the results for Memory access time against local cache size and number of clients for the FIFO algorithm. FIFO is a basic queue implementation and in known to perform poorly as compared to other cache replacement algorithm. N-block replacement improves the memory access time than single block, but as compared to other replacement algorithms, this is still a poor performance.

Figure 3: FIFO- Local Cache Size vs. Memory Access Time

Figure 3 shows the graph of single block vs. N-block replacement for the memory access time over the local cache size. For low cache size, N-block replacement reduces memory access time considerably, but for large cache sizes, the difference drops as it can be seen in the graph.

Figure 4 shows the graph of single block vs. N-block replacement for the memory access time over the local cache size. N-block replacement for varied number of clients reduces the access time with a large difference for small number of clients. However, with increase in number clients, the access time remains almost the same. Single block replacement shows an exponentially decreasing curve for small number of clients and becomes steady with increase in number of clients.

5.1.2 Least Frequently Used
Figure 5 shows the graph of single block vs. N-block replacement for the memory access time over the local cache size. For lower cache sizes, N-block replacement reduces the access time almost by half and hence improves the performance of the cache. But, as it can be seen in the graph, as the cache size increases the performance deteriorates in comparison with single block. This is one of the disadvantages of LFU replacement policy, if the access pattern changes too often, the performance takes a hit. And, since N-block has patterns that are set of blocks appearing at random.

Figure 6 shows the graph of single block vs. N-block replacement for memory access time over number of clients. When the number of clients is low, the time decreases by a large factor, and as the number of clients increase, N-block replacement still performs better than the single block replacement with less significant difference. As seen the graph, the N-block replacement has an almost constant memory access time as opposed to single block which decreases steadily.

5.1.3 Least Recently Used

Figure 7 shows the graph of single block vs. N-block replacement for the memory access time over the local cache size. For lower cache sizes, N-block replacement reduces the access time by more than half and hence improves the performance of the cache. But, as it can be seen in the graph, as the cache size increases the difference between the two starts to drop. LRU is known to perform better than the FIFO and Random replacement policies.
Figure 8: LRU- Number of Clients vs. Memory Access Time

Figure 8 shows the graph of single block vs. N-block replacement for the memory access time over the number of clients. N-block replacement by itself does not demonstrate the desired result to improve the performance of the cache, but it still performs better than single block. With increase in number of clients, the availability of the data in neighbors also increase, the single block replacement curve looks exponentially decreasing for low number of clients and decreases steadily for high number of clients.

5.1.4 NChance

Figure 9: NChance- Local Cache Size vs. Memory Access Time

Figure 9 shows the graph of single block vs. N-block replacement for the memory access time over the local cache size. For lower cache sizes, N-block replacement reduces the access time by more than half and hence improves the performance of the cache. But, as it can be seen in the graph, as the cache size increases the difference between the two starts to drop. N-block replacement performs better for all cache sizes, but not exponentially decreasing like single block. In both cases, the curve flattens out a certain cache size and remains constant. NChance forwarding retains the singlet which makes it a better performer than the other replacement strategies, but has an additional overhead of sending the data to neighbor. This could be a significant overhead for large number of data requests.

Figure 10: NChance- Number of Clients vs. Memory Access Time

Figure 10 shows the graph of single block vs. N-block replacement for the memory access time over the number of clients. N-block replacement by itself does not demonstrate the desired result to improve the performance of the cache, but it still performs better than single block. With increase in number of clients, the availability of the data in neighbors also increase, the single block replacement curve looks exponentially decreasing for low number of clients and decreases steadily for high number of clients.

5.2 Miss Rate

The goal is to reduce the cache miss rate to improve the performance of the cache. This should be done such that it does not cost too much of access time. Larger cache sizes would reduce the miss rate, but it will increase the access time and the resources required to maintain a large cache is very expensive. Increase in number of clients reduces miss rate to some extent, but miss rate reaches 0 for larger cache sizes only.

5.2.1 First In First Out

Figure 11 shows the graph of single block vs. N-block replacement for the cache miss rate over the local cache size. For low cache sizes, N-block replacement reduces miss rate by a very large factor. The behavior converges for large cache sizes.

Figure 12 shows the graph of single block vs. N-block replacement for the cache miss rate over number of clients. Increasing number of clients reduces the miss rate almost linearly. N-Block replacement has fuzzy curve as it can be seen in the figure. This is because, the miss rate reduces when there is replacement of n-blocks and increases when there single data blocks. This largely depends on where the set of blocks are situated in the incoming requests, meaning the access pattern of the data.
5.2.2 Least Frequently Used

Figure 13 shows the graph of single block vs. N-block replacement for the cache miss rate over the local cache size. For lower cache sizes, N-block replacement reduces the miss rate and hence improves the performance of the cache. As it can be seen in the graph as the cache size increases, the N-block replacement behavior is very similar to single block replacement. This is because with larger cache sizes, the data blocks tend to reside in the cache.

5.2.3 Least Recently Used

Figure 14 shows the graph of single block vs. N-block replacement for miss rate over number of clients. When the number of clients is low, the miss rate decreases by a large factor, and as the number of clients increase, N-block replacement has varied levels of good and bad performance as compared to single block replacement. This behavior is because the N-block replacement makes space as the incoming requests arrive, the miss rate decreases when there is N-block replacement and increases when there is single block replacement.
replacement for the cache miss rate over the local cache size. For low cache sizes, N-block replacement reduces miss rate by a very large factor. The behavior converges for large cache sizes.

Figure 16: LRU- Number of Clients vs. Miss Rate

Figure 16 shows the graph of single block vs. N-block replacement for the cache miss rate over number of clients. Increasing number of clients reduces the miss rate almost linearly. N-Block replacement has fuzzy curve as it can be seen in the figure. This is because, the miss rate reduces when there is replacement of n-blocks and increases when there single data blocks. This largely depends on where the set of blocks are situated in the incoming requests, meaning the access pattern of the data.

5.2.4 NChance

Figure 17: NChance- Local Cache Size vs. Miss Rate

Figure 17 shows the graph of single block vs. N-block replacement for the cache miss rate over the local cache size. For low cache sizes, N-block replacement reduces miss rate by a very large factor. The behavior converges for large cache sizes.

Figure 18: NChance- Number of Clients vs. Miss Rate

Figure 18 shows the graph of single block vs. N-block replacement for the cache miss rate over number of clients. Increasing number of clients reduces the miss rate almost linearly. N-Block replacement has fuzzy curve as it can be seen in the figure. This is because, the miss rate reduces when there is replacement of n-blocks and increases when there single data blocks. This largely depends on where the set of blocks are situated in the incoming requests, meaning the access pattern of the data.

5.3 Hit Rate

The goal is to increase the cache hit rate to improve the performance of the cache. This should be done such that it does not cost too much of access time. Larger cache sizes increases the hit rate, but it will increase the access time and the resources required to maintain a large cache is very expensive. We discuss two separate hit rates as discussed in section 3.2. The end goal is to maximize both hit rates without causing performance degradation in other metrics. Increase in number of clients improves neighbor hit rate to a certain extent, but has no effect on the local hit rate. Each algorithm shows very similar behavior but due to their own properties. We discuss each of them in the following section.

5.3.1 First In First Out

Figure 19 shows the graph of single block vs. N-block replacement for the local hit rate over the local cache size. Single block replacement increases the hit rate in a slow linear fashion with increase in cache size. For very low cache sizes, N-block replacement increases the hit rate exponentially as compared to single block replacement. But, as it can be seen in the graph, as the cache size increases the N-block replacement increases gradually and becomes steady at about the cache size of 512K. This is because with larger cache sizes, there is enough space for incoming data blocks, and replacing N blocks at a time frees the cache space as much as single block.

Figure 20 shows the graph of single block vs. N-block replacement for the neighbor hit rate over the local cache size. Single block replacement increases the hit rate in an exponentially for low cache sizes and tops out at high cache sizes.
sizes. N-block replacement increases the hit rate by a large factor for low cache sizes, and decreases for moderate to high cache sizes. With increase in cache size, N-block replacement increases the availability of space in the local cache, due to which the clients can serve for the data requests from their own cache. This reduces fetching data from neighbors, which naturally decreases the neighbor hit rate and increases the local hit rate. One disadvantage of the FIFO replacement policy is that it takes no account of recency or frequency of usage, the data blocks are evicted the way they are read. This is not a desirable property for N-block replacements since the data blocks could be evicted by another client even before they are read completely by the current one. This has an adverse affect on the hit rate.

Figure 21 shows the graph of single block vs. N-block replacement for the local hit rate over number of clients. Single block replacement has a constant local hit rate for varied number of clients where as the curve for the N-block replacement varies with number of clients. This behavior is due to the pattern of incoming blocks requested, every time there is a N-block replacement, the local hit rate increases due to more space for a certain period of time, if there is a single block replacement the local hit rate drops. This depends on structure of incoming request blocks. The FIFO replacement policy takes no account of recency or frequency of usage, the data blocks are evicted the way they are read. This is not a desirable property for N-block replacements since the data blocks could be evicted by another client even before they are read completely by the current one.

Figure 22 shows the graph of single block vs. N-block replacement for the neighbor hit rate over number of clients. Single block replacement increases the hit rate in an almost linear way with increase in number of clients. This is desired behavior of the hit rate metric. While the N-block replacement improves the performance of cache by increasing the hit rate by a large factor for small number of clients, there
is an increase-decrease pattern with increase in number of clients, this is not a desired behavior. This is because there is a direct impact on memory performance with irregular behavior of hit rate. The goal is to have steadily increasing cache hit rate without compromising on the memory performance.

5.3.2 Least Frequently Used

Figure 23: LFU- Local Cache Size vs. Local Hit rate

Figure 23 shows the graph of single block vs. N-block replacement for the local hit rate over the local cache size. For very low cache sizes, N-block replacement and single block replacement does not behave very different. But, as it can be seen in the graph, as the cache size increases the N-block replacement increases the cache hit rate. This is because with larger cache sizes, there is enough space for incoming data blocks, and replacing N blocks at a time frees the cache space, which in turn reduces the eviction. This results in data block residing in the cache for longer periods.

Figure 24: LFU- Local Cache Size vs. Neighbor Hit Rate

Figure 24 shows the graph of single block vs. N-block replacement for the neighbor hit rate over the local cache size. For very low cache sizes, N-block replacement shows an increase in the neighbor hit rate. But, as it can be seen in the graph, as the cache size increases the N-block replacement starts to show results same as single block. For moderate to higher sized caches, N-block replacement shows a poor performance as compared to single block. This is because with larger cache sizes, there is enough space for incoming data blocks in local cache, and replacing N blocks at a time won’t have any effect on neighbor hit rate since everything that is needed is mostly in client’s local cache. But in single block replacement, since the eviction is done one block at a time, finding data in neighbor has a higher probability than holding on to data blocks for longer periods.

Figure 25: LFU- Number of Clients vs. Local Hit Rate

Figure 25 shows the graph of single block vs. N-block replacement for local hit over number of clients. The local hit rate increases by a large factor for N-block replacement. Single block replacement has a constant local hit rate for varied number of clients where as the curve for the N-block replacement varies with number of clients. This behavior is due to the pattern of incoming blocks requested, every time there is a N-block replacement, the local hit rate increases due to more space for a certain period of time, if there is a single block replacement the local hit rate drops. This depends on structure of incoming request blocks.

Figure 26 shows the graph of single block vs. N-block replacement for neighbor hit rate over number of clients. The neighbor hit rate increases by a large factor for N-block replacement. Single block replacement has an increasing neighbor hit rate for varied number of clients since the number of neighbors available to look up also increases. N-Block replacement shows a better performance in terms of increasing the hit rate, but the performance does not improve beyond a certain number of clients, there is a periodic increase and decrease.

5.3.3 Least Recently Used

Figure 27: LFU- Number of Clients vs. Neighbor Hit Rate

Figure 27 shows the graph of single block vs. N-block replacement for neighbor hit over number of clients. The neighbor hit rate increases by a large factor for N-block replacement. Single block replacement has an increasing neighbor hit rate for varied number of clients since the number of neighbors available to look up also increases. N-Block replacement shows a better performance in terms of increasing the hit rate, but the performance does not improve beyond a certain number of clients, there is a periodic increase and decrease.
Figure 26: LFU- Number of Clients vs. Neighbor Hit Rate

Figure 27: LRU- Local Cache Size vs. Local Hit Rate

Figure 27 shows the graph of single block vs. N-block replacement for the local hit rate over the local cache size. Single block replacement increases the hit rate linearly with increase in cache size. For very low cache sizes, N-block replacement increases the hit rate exponentially as compared to single block replacement. But, as it can be seen in the graph, as the cache size increases the N-block replacement increases gradually. This is because with larger cache sizes, there is enough space for incoming data blocks, and replacing N blocks at a time frees the cache space, which in turn reduces the eviction. This results in data block residing in the cache for longer periods.

Figure 28: LRU- Local Cache Size vs. Neighbor Hit Rate

Figure 28 shows the graph of single block vs. N-block replacement for the neighbor hit rate over the local cache size. Single block replacement increases the hit rate in an exponentially for low cache sizes and tops out at high cache sizes. N-block replacement increases the hit rate by a large factor for low cache sizes, and decreases for moderate to high cache sizes. With increase in cache size, N-block replacement increases the availability of space in the local cache, due to which the clients can serve for the data requests from their own cache. This reduces fetching data from neighbors, which decreases the neighbor hit rate and increases the local hit rate.

Figure 29: LRU- Number of Clients vs. Local Hit Rate

Figure 29 shows the graph of single block vs. N-block replacement for local hit rate over number of clients. The local hit rate increases by a large factor for N-block replacement. Single block replacement has a constant local hit rate for varied number of clients where as the curve for the N-block replacement varies with number of clients. This behavior is due to the pattern of incoming blocks requested, every time there is a N-block replacement, the local hit rate increases due to more space for a certain period of time, if there is
a single block replacement the local hit rate drops. This depends on structure of incoming request blocks.

Figure 30: LRU- Number of Clients vs. Neighbor Hit Rate

Figure 30 shows the graph of single block vs. N-block replacement for neighbor hit rate over number of clients. The neighbor hit rate increases by a large factor for N-block replacement. Single block replacement has an increasing neighbor hit rate for varied number of clients since the number of neighbors available to look up also increases. N-Block replacement shows a better performance in terms of increasing the hit rate, but the performance is steady and has very minimum improvement for large number of clients.

5.3.4 NChance

Figure 31: NChance- Local Cache Size vs. Local Hit Rate

Figure 31 shows the graph of single block vs. N-block replacement for the local hit rate over the local cache size. Single block replacement increases the hit rate very slowly and linearly with increase in cache size. For very low cache sizes, N-block replacement increases the hit rate exponentially as compared to single block replacement. But, as it can be seen in the graph, as the cache size increases the N-block replacement increases at the same rate and tops out at about the cache size of 1024K. This is because with larger cache sizes, there is enough space for incoming data blocks, and replacing N blocks at a time frees the cache space, which in turn reduces the eviction. This results in data block residing in the cache for longer periods.

Figure 32: NChance- Local Cache Size vs. Neighbor Hit Rate

Figure 32 shows the graph of single block vs. N-block replacement for the neighbor hit rate over the local cache size. Single block replacement increases the hit rate in an exponentially for low cache sizes and tops out at high cache sizes. N-block replacement increases the hit rate by a large factor for low cache sizes, and decreases for moderate to high cache sizes. With increase in cache size, N-block replacement increases the availability of space in the local cache, due to which the clients can serve for the data requests from their own cache. This reduces fetching data from neighbors, which decreases the neighbor hit rate and increases the local hit rate.

Figure 33: NChance- Number of Clients vs. Local Hit Rate

Figure 33 shows the graph of single block vs. N-block replacement for local hit rate over number of clients. The local
hit rate increases by a large factor for N-block replacement. Single block replacement has a constant local hit rate for varied number of clients where as the curve for the N-block replacement varies with number of clients. This behavior is due to the pattern of incoming blocks requested, every time there is a N-block replacement, the local hit rate increases due to more space for a certain period of time, if there is a single block replacement the local hit rate drops. This depends on structure of incoming request blocks.

Figure 34 shows the graph of single block vs. N-block replacement for neighbor hit rate over number of clients. The neighbor hit rate increases by a large factor for N-block replacement. Single block replacement has an increasing neighbor hit rate for varied number of clients since the number of neighbors available to look up also increases. N-Block replacement shows a better performance in terms of increasing the hit rate, but the performance is steady and has very minimum improvement for large number of clients, while the goal is to maximize the hit rate with increase in number of clients.

**5.4 Disk Access Rate**

The goal is to reduce the number of disk accesses. The access time increases as the number of disk accesses increases. The whole idea of having a well performing cache is to reduce the number of disk accesses. In our experiments, we found out that N-block replacements reduce the disk access rate for low cache sizes and small number of clients. For large cache and moderate to high to clients, the disk access rate drops to almost zero.

**5.4.1 First In First Out**

Figure 35 shows the graph of single block vs. N-block replacement for the disk access rate over the local cache size. For lower cache sizes, N-block replacement reduces the disk access rate almost by half and hence improves the performance of the cache. But, as it can be seen in the graph, as the cache size increases the N-block replacement and single block replacement both behave similarly and the access rate converges to 0. This is because with larger cache sizes, the data blocks tend to reside in the cache, which is the same effect for both single and N-block replacements.

Figure 36 shows the graph of single block vs. N-block replacement for disk access rate over number of clients. When the number of clients is low, the disk access rate decreases by a large factor, and as the number of clients increase, N-block replacement and single block replacement has the same results. This is because the disk access rate converges to 0 irrespective of replacement strategies due to high availability of the requested data in one or more clients.

**5.4.2 Least Frequently Used**

Figure 37 shows the graph of single block vs. N-block replacement for the disk access rate over the local cache size. For lower cache sizes, N-block replacement reduces the disk access rate almost by 40% and hence improves the performance of the cache. But, as it can be seen in the graph, as the cache size increases the N-block replacement and single block replacement both behave similarly and the access rate converges to 0. This is because with larger cache sizes, the data blocks tend to reside in the cache, which is the same effect for both single and N-block replacements.
Figure 37: LFU- Local Cache Size vs. Disk Access Rate

Figure 38: LFU- Number of Clients vs. Disk Access Rate

Figure 38 shows the graph of single block vs. N-block replacement for disk access rate over number of clients. When the number of clients is low, the disk access rate decreases by a large factor, and as the number of clients increase, N-block replacement and single block replacement has the same results. This is because the disk access rate converges to 0 irrespective of replacement strategies due to high availability of the requested data in one or more clients.

5.4.3 Least Recently Used

Figure 39 shows the graph of single block vs. N-block replacement for the disk access rate over the local cache size. For lower cache sizes, N-block replacement reduces the disk access rate almost by half and hence improves the performance of the cache. But, as it can be seen in the graph, as the cache size increases the N-block replacement and single block replacement both behave similarly and the access rate converges to 0. This is because with larger cache sizes, the data blocks tend to reside in the cache, which is the same effect for both single and N-block replacements.

5.4.4 NChance

Figure 40 shows the graph of single block vs. N-block replacement for disk access rate over number of clients. When the number of clients is low, the disk access rate decreases by a large factor, and as the number of clients increase, N-block replacement and single block replacement has the same results. This is because the disk access rate converges to 0 irrespective of replacement strategies due to high availability of the requested data in one or more clients.

Figure 41 shows the graph of single block vs. N-block replacement for the disk access rate over the local cache size. For lower cache sizes, N-block replacement reduces the disk access rate almost by half and hence improves the performance of the cache. But, as it can be seen in the graph, as the cache size increases the N-block replacement and single block replacement both behave similarly and the access rate converges to 0. This is because with larger cache sizes, the data blocks tend to reside in the cache, which is the same effect for both single and N-block replacements.
5.5 Comparison of Set of Block Occurrence

This section shows the results for comparison of different concentration of set of blocks in input requests. We performed experiments for all the algorithms of interest. This section shows the results for NChance forwarding algorithm, which has a better performance as compared to the other three. In our experiments, we have used 5%, 10%, and 15% of total data blocks as set of blocks. The results show that the higher the concentration of set of blocks, N-block replacement shows improvement in the performance of the cache. This is the expected behavior, when there are more set of blocks, every time we replace a set of blocks, this makes more space in cache which in turn reduces the calls to replacement algorithms, and also makes the existing data blocks stay in the cache for longer periods.

Figure 43 shows the graph of memory access time over the local cache size for varying percentage of occurrence of set of blocks. As the percentage of set of blocks increase in the input data, the local cache size increase decreases the memory access time in N-block replacement strategies. This behavior is expected since more number of set of blocks means the eviction of N-blocks is often which frees up the local cache. Our experiments, having 15% of data as a set of blocks shows the most desirable results.

Figure 44 shows the graph of memory access time over the local cache size for varying percentage of occurrence of set of blocks. As the percentage of set of blocks increase in the input data, with local cache size increase, the local hit rate in N-block replacement strategies has a better performance. For larger cache sizes, the increase in local hit rate becomes steady. This behavior is expected since more number of set of blocks means the eviction of N-blocks is often which frees up the local cache. This has an impact on smaller cache
sizes, but in larger cache sizes this replacement makes the performance of cache a constant since the space freed up by eviction is very small as compared to the cache size itself. Our experiments, having 15% of data as a set of blocks shows the most desirable results.

Figure 45: Local Cache Size vs. Neighbor Hit Rate

Figure 45 shows the graph of neighbor hit rate over the local cache size for varying percentage of occurrence of set of blocks. For very low cache sizes, N-block replacement shows an increase in the neighbor hit rate. But, as it can be seen in the graph, as the cache size increases the N-block replacement starts to show results same as single block. For moderate to higher sized caches, N-block replacement shows a poor performance as compared to single block. This is because with larger cache sizes, there is enough space for incoming data blocks in local cache, and replacing N blocks at a time won’t have any effect on neighbor hit rate since everything that is needed is mostly in client’s local cache. But in single block replacement, since the eviction is done one block at a time, finding data in neighbor has a higher probability than holding on to data blocks for longer periods. Our experiments shows the most desirable results for single block replacement.

Figure 46: Local Cache Size vs. Miss rate

Figure 46 shows the graph of miss rate over the local cache size for varying percentage of occurrence of set of blocks. The miss rate decreases with increase in the local cache size of all inputs. Results show that the miss rate is the least for highest percentage of N-blocks. For high cache sizes, the behavior is same irrespective of N-block concentration.

Figure 47: Local Cache Size vs. Disk Access rate

Figure 47 shows the graph of disk access rate over the local cache size for varying percentage of occurrence of set of blocks. The disk access rate decreases with increase in the local cache size of all inputs. Results show that the miss rate is the least for highest percentage of N-blocks. For high cache sizes, the behavior is same irrespective of N-block concentration.

5.6 NChance vs. NChance-K

This section shows the results for NChance and NChance-K algorithms. NChance-K is our version of modified NChance greedy forwarding algorithm as described in section 3.3. The experimental setup for this was dynamically varying the value of N depending on the frequency of occurrence of a particular data block. The value of N used was 3, 4, and 5 depending on the occurrence threshold. Our results showed that there no significant improvement on any of the cache performance metrics with the modified NChance forwarding algorithm. The difference in results between NChance and NChance-K was negligible. The comparison of these two algorithms are shown below and the behavior is exactly same as NChance as described in the above sections.

Figure 48: Local Cache size vs. Memory access time for NChance and NChance-K
Figure 49: Local Cache size vs. Miss Rate for NChance and NChance-K

Figure 50: Local Cache size vs. Local Hit Rate for NChance and NChance-K

Figure 51: Local Cache size vs. Neighbor Hit Rate for NChance and NChance-K

Figure 52: Local Cache size vs. Disk Access Rate for NChance and NChance-K

5.7 Algorithm Comparison

This section shows the results for the comparison of all the algorithms for single block vs. N-block replacement. The experimental parameters are kept same for both cases, and the replacement strategy is changed.

Figure 53: Local Cache size vs. Memory access time for single block and N-block

Figure 54 shows the graph of memory access time over the local cache size. For small cache sizes, N-block replacement reduces the memory access time. As the cache size increases, the behavior is similar to single block replacement. This is because the access time decreases since the number of times replacement algorithm called drops because of the large cache size.

Figure 55: Local Cache size vs. Miss Rate for single block and N-block

Figure 54 shows the graph of miss rate over the local cache size. For small cache sizes, N-block replacement reduces the cache miss because every time the replacement algorithm is called, replacing N blocks frees us more space which also means whatever is already there in the cache stays there for longer. When the cache is bigger, the probability of finding a data block locally or in the neighbor is high because of availability of space to hold on to data blocks for longer periods of time, this is true for both replacement strategies.

Figure 55: Local Cache size vs. Local Hit Rate for single block and N-block

Figure 55 shows the graph of local hit rate over the local cache size. Single block replacement increases the local hit rate linearly whereas the N-block replacement improves the hit rate by a large factor for small and moderate cache sizes and then flattens out after that.
Figure 56: Local Cache size vs. Neighbor Hit Rate for single block and N-block

Figure 56 shows the graph of neighbor hit rate over the local cache size. Single block replacement increases the neighbor hit rate with the size of the cache and tops out for large cache sizes. The N-block replacement improves the hit rate for small cache sizes only. For moderate and large cache sizes N-Block replacement shows a poor performance than the single block, this is because the N-block replacement finds data blocks locally since the freed up space from N-block replacement holds on to the data blocks for a longer time.

Figure 57: Local Cache size vs. Disk Access Rate for single block and N-block

Figure 57 shows the graph of disk access rate over the local cache size. For small cache sizes, N-block replacement reduces the number of disk access by a large factor because every time the replacement algorithm is called, replacing N blocks frees us more space which also means whatever is already there in the cache stays there for longer. When the cache is bigger, the disk accesses reduce because of the availability of a large space, this is true for both replacement strategies.

6. FUTURE WORK

- Simulation study can be extended to use data files instead of data blocks. This will enable us to study the behavior of cache with large files in N-block replacement domain.
- Simulation study can be extended to implement several modified versions of the FIFO, LRU, and LFU algorithms to study the effect of N-block replacement in these strategies. Some of them are described in [4].

7. CONCLUSION

N-Block replacement is suitable if input data requests contains blocks of data that are always needed together. We observed the following for N-Block replacement:

- Improves the performance of cache for low local cache sizes with varied number of clients.
- There is a steep decrease in memory access time, disk access rate, and miss rate.
- There is an increase in cache hits, either local or neighbor or both to some degree in certain cases.
- As the percentage of block data (N-Blocks) increases in the data requests, the cache performance is better.
- For low local cache sizes, shows a significant improvement in cache performance for all the metrics.
- For high local cache sizes, behaves similar to single block replacement.
- For smaller number of clients, and low cache sizes, shows a significant improvement in cache performance for all the metrics.
- For higher number of clients, behaves similar to single block replacements irrespective of cache sizes.

There is no change in performance of cache for varied server cache size, both for single and N-block replacement. Of the four algorithms evaluated, N-Chance replacement algorithm performs better than the other three.

8. REFERENCES