Cooperative Caching With Adaptive Asynchronous Prefetching

Niket Mhatre [1], Ms.Mona Mulchandani [2] Ms.Swara Pampatwar [3], Ms.Mayuri Chawala [4]
Department of Computer Science, Rochester Institute Of Technology, NewYork USA
ndm5134@rit.edu

***Abstract*—**
This project presents integration and simulation road map of an adaptive asynchronous pre-fetching (AAP) schema into the hint based cooperative caching. Our contribution involves improving the hint-based algorithm to accommodate AAP pre-fetching scheme that reduces cache wastage by improving LRU policy and cache pollution by adapting the degree of pre-fetch. The two best pre-fetching algorithms have been considered for comparison, which are – One block look ahead with linear aggressive pre-fetching and an interval and size prediction-by-partial-match (IS_PPM). To measure the success of our proposed integration we have simulated them and provided a comparison analysis

***Keywords*—** Adaptive ,Asynchronous, Prefetching, hint-based, cooperative cache.

I. INTRODUCTION

Today's distributed file systems use a three-level memory hierarchy namely server disk, server cache and client cache. Server and client cache play an important part in distributed file systems because they reduce frequent accesses to the much slower server disk. The server cache-hits limit the performance of a distributed file system because any local cache-miss requires access to the server cache and any server cache-miss leads to accessing of the server disk. Previous research [6] has shown that increasing the client and server cache does reduce disk access but is not as effective as it appears and is also really expensive. Overall performance can be improved by distributing additional memory across clients and introducing a logical layer into the memory hierarchy of distributed file system [7]. This logical layer is called the cooperative cache, which allows clients to access cached blocks from other client's memory.

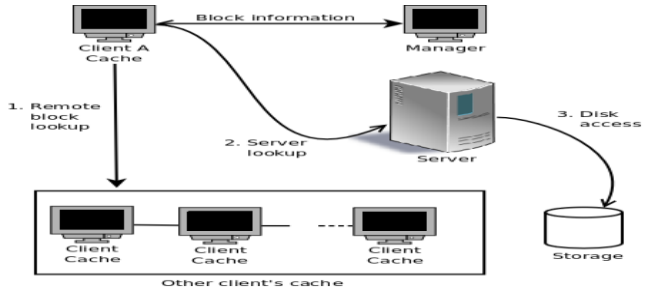

Fig. 1 Overview of the cooperative caching system. Client A requests block information from the manager. Then a client performs a remote lookup into the other client's caches (1). If a block is not found in the cooperative cache then (2) the server lookup is performed. In case of the server cache-miss, (3) a block is fetched from the storage device.

II. OVERVIEW OF COOPERATIVE CACHING

Fig. 1 shows a typical architecture of the cooperative caching system. Cooperative caching architecture involves three main components namely clients, servers and a manager. The logical layer of cooperative cache is comprised of client caches and each client stores a block into the cache for itself as well as for other clients. Whenever a client requests a block, it will try to locate it into its own local cache and if it's not found, it attempts to fetch it from the cooperative cache. Similarly, whenever a client replaces an old block, it may either be forwarded to other clients for storage or discarded, depending upon the algorithm. The cooperative caching schema must provide this lookup-and-forwarding mechanism for managing the blocks. Also, the manager may provide coordination among the clients for locating and forwarding the blocks. Various algorithms have been presented with different lookup, forwarding and coordination strategies. Some of the well-known algorithms are N-chance [6] by Dahlin et al., Hint based caching [7] by Sarkar et al. and RoundRobin [1] by Anderson et al.

Pre-fetching

Although, the above proposed algorithms aim at increasing overall throughput of the system; there is still a vast scope for improvements. The cooperative cache offers a huge amount of aggregate memory that is not used to its potential. In most cases we can find blocks that are not accessed for many hours. Using pre-fetching technique, performance can be improved by replacing these unused blocks with those that might be requested in the future. Pre-fetching is commonly used along with cooperative caching to gain additional performance.

Pre-fetching algorithms may be sequential or random based on the access pattern they follow. Sequential is more popular of the two since files are generally accessed in a sequential manner [2]. Sequential pre-fetching algorithms are further classified into four classes: fixed synchronous, fixed

asynchronous, adaptive synchronous and adaptive asynchronous [4]. These classifications are based on whether the algorithm has a fixed or an adaptive degree of pre-fetch and also if pre-fetching is synchronous or asynchronous.

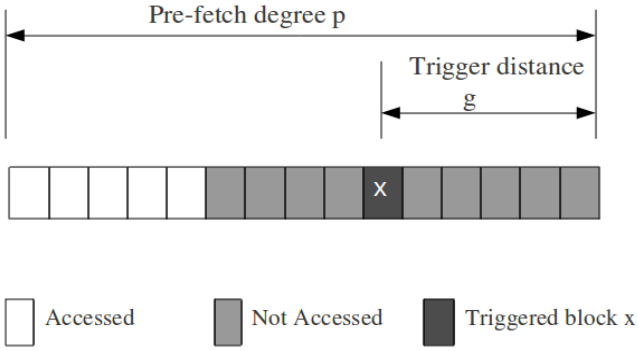

Fig 2. Asynchronous pre-fetching. [4]

With reference to Figure 2, P is current pre-fetch set, G is triggered distance and X is triggered block at distance of G from the end of pre-fetch set. Pre-fetching is said to be synchronous if p number of blocks are read by the client when a cache miss occurs on block x. On the other hand, in asynchronous pre-fetching, p blocks are read when there's a cache hit on block x. Asynchronous pre-fetching allows us to stay ahead of a read request, and hence reduces the cache miss ratio. Figure 2 and Algorithm 1 gives an overview of asynchronous pre-fetching.

Sequential pre-fetching algorithms have two main issues, namely cache pollution and wastage of pre-fetched blocks [4]. Cache pollution occurs, when pre-fetched blocks replace more useful blocks. It can happen with an aggressive pre-fetching policy. Another problem is cache wastage, where pre-fetched blocks are evicted from the cache before they are used. This is a more serious problem. It not only overloads the cache by multiple pre-fetching of the same blocks but also increases network bandwidth usage unnecessarily.

III. RELATED WORK

In this section we will look at a cooperative caching algorithm and pre-fetching techniques that will give us some background information required for presenting the proposed idea. We will first explain basics of hint-based cooperative caching and then glance into the pre-fetching algorithms.

Cooperative caching algorithm: Hint-Based caching

Sarkar et al. proposed an algorithm using "hints" as means locating a block. Hints are a probable location of a block and easy to maintain as opposed to more accurate information in the traditional tightly controlled approach [hint]. This is because maintaining facts requires constant synchronization between the clients and the manager. They also help reduce the communication overhead with the manager. Hints need be accurate to achieve a better performance of the system. Hence the manager obtains the hints from the last client to request the file, as that client is most likely to have accurate hints. Tracking each copy of a block in an entire cooperative cache is prohibitively expensive. Hence the algorithm keeps track of only the first copy to be cached called as the master copy.

A client starts a block lookup by checking its own local cache. If the block is not found in the local cache then the client obtains hints for the blocks from the manager. After getting hints from the manager, the client forwards the request to an appropriate client. The client forwards the request to the server in case of inaccurate hints. While adding a new block into the local cache, the algorithm uses the least recently used (LRU) replacement policy. In this algorithm, forwarding of a replaced block is more targeted with the help of a best-guess replacement technique [hint]. In this technique each client maintains a sorted list that contains oldest block age of other clients. This list helps the client to identify the target. While forwarding the block to another client, two clients update this list by exchanging an oldest block's age information. To reduce an overhead, the algorithm forwards only the master copy of the block and the other copies are discarded. When master copy is forwarded to the client, it may keep the copy or discard it based on its own oldest block age. At a given time an oldest block list may not be up-to-date. Due to this incomplete information, the client may discard a forwarded master copy, in which case, it's forwarded to another cache on the server called as discarded cache [hint]. All the discarded master copies are forwarded to this cache for avoiding an expensive disk access.

In this project, the hint-based cooperative caching system is used since it is more efficient than the other cooperative caching systems out there [7].

Pre-fetching algorithms:

One block lookahead with linear aggressive pre-fetching:

One block lookahead (OBA) is one the most commonly used pre-fetching algorithm. The idea of the algorithm is simple; pre-fetch the block following the one requested. Dahlgren et al. extended an idea of OBA by proposing Mth block lookahead [3]. In this algorithm, M blocks are pre-fetched instead of just one. This algorithm takes advantage of the fact that blocks are accessed sequentially. Cortes et al. proposed linear aggressive pre-fetching algorithm [2] for the cooperative caching system using an extended version of one block lookahead. In this algorithm, the degree of pre-fetch (the number of blocks per pre-fetch) is fixed. In this paper the author takes advantage of the huge amount of memory offered by the cooperative caching system.

Interval and size prediction-by-partial-match

Cortes et al. proposed IS_PPM [2] algorithm for pre-fetching using an access pattern of the file. Their algorithm is based on the concept that files are accessed sequentially or strided pattern [reference] as shown in a Figure 3. The algorithm is derived from prediction-by-partial-match (PPM) [5], proposed by Vitter et al.

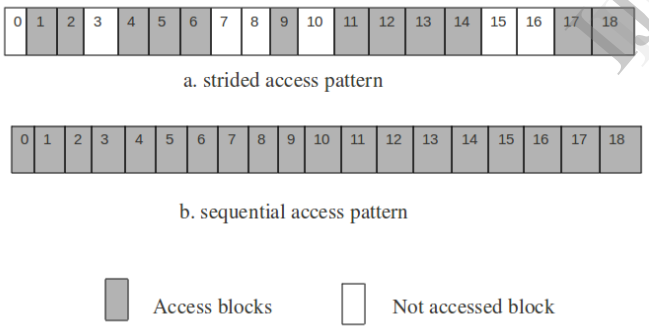

Fig 3: File access pattern.

The IS_PPM is based on the Markov model, which states that the probability distribution of the future states is dependent upon the present and the past states [5]. Using this model, they construct a graph for predicting an access pattern. Each node in the graph consists of the size of request and an offset interval. The offset interval is the difference between the first block of the current and previous requests. Whenever new request is made, the system computes the size and an offset interval between the current and the previous request. Using this information graph is searched for the node with matching information. Once the node is found, the system follows the most updated link and extracts the interval and the size of the future request from the linked node. In case the node is not found, then the OBA algorithm is used for pre-fetching the request.

Adaptive asynchronous pre-fetching

Gill et al. have proposed an adaptive multi-stream pre-fetching (AMP) algorithm for shared cache. The aim of an algorithm is to provide online optimization of pre-fetched data and reduce problem associated with pre-fetching. Adaptive asynchronous pre-fetching algorithm is an asynchronous and adaptive algorithm and is an application of AMP in cooperative cache [reference].It adapts the value of perfected degree p and triggered distance g. As per the algorithm, asynchronous pre-fetching is activated as soon as P is greater than the threshold (t). The threshold is set to the value of 4. A block at distance t/2 from the end of perfected set is used as a triggered block. Whenever there is a cache-hit on a triggered block, p sets of blocks are pre-fetched and the triggered block position is reset.

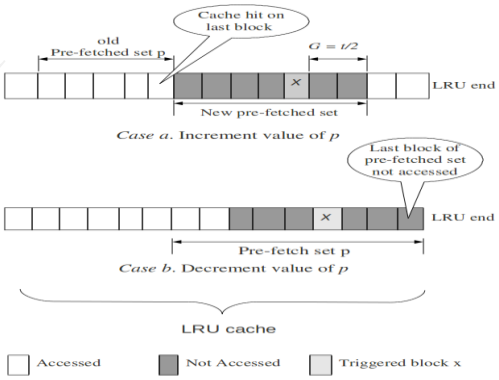

Fig 4: File access pattern

Algorithm adapts the values of p and g to reduce the pre-fetched wastage and cache pollution. For this, the values of p and g are incremented or decremented as follows. (See Figure 4) If the last block of the current pre-fetched set is not accessed then it is an indication that the current value of p is too high. We then decrement the value of p. In this case, the last block from the pre-fetched set will move to the end of the LRU queue without being accessed. We can give another chance to such a block by moving it one position ahead towards the head of the LRU queue. Whenever the last block in the current pre-fetched set is hit, we increment the values of p and g.

IV.PROBLEM STATEMENT

One of the main goals of cooperative caching is to maximize throughput while minimizing the cost of cache maintenance.

The hint-based cooperative caching achieves this optimality by reducing the workload on the manager. Currently, a block is fetched into the cache on demand. When a block is requested for the first time, on every block request cache miss is associated. We can reduce these cache misses by pre-fetching the future request. The cooperative cache offers a huge amount of aggregate memory, which can be well utilized. In many cases we can find the blocks in the cooperative cache that are not accessed for hours. Performance can be improved by replacing these unused blocks with more useful ones. Most of the pre-fetching algorithms suffer from the problems of pre-fetch wastage and cache pollution due to fixed degree of pre-fetch. We can reduce these problems by introducing an adaptive degree of pre-fetch and improved LRU policy by making it aware of the pre-fetched blocks.

Hint-based system implementation

We have implemented our system in a simulated environment. The hint-based system is our base implementation. This section explains the details of the simulation environment and the various parameters used.

The simulation environment is implemented using Java. The system is having five main objects namely Simulator, Server, Client, Disk and Manager. The disk object emulates the storage device. It is responsible for set of files and generating the blocks. The files are represented using class objects with different file sizes. The server objects are responsible for disk object, server cache and discarded cache. Each disk read takes m milliseconds to access a file block, where m is a constant value set through the configuration file. The client objects are responsible for simulating the read requests and managing the local cache. Each remote block access time takes n milliseconds where n is constant. A remote access time includes round trip time. It is the total of time taken by a client to send the message to the remote client and time to transfer the block. The server cache access time is larger than client cache access time. The manager object keeps track of the set of hints for the blocks of a file. In order to keep hints accurate, manager updates location of the hint to the last client who has opened the file. The simulator object reads the configuration file and initiates simulation as per this file. It is also responsible for output graphs. The parameters we consider for our simulation are mentioned below. These are maintained by the configuration files.

NO_OF_SERVER : Total number of servers
SERVER_DISK_FILES : Total number of files each server should have. (Files are simulated with random file size)
CLIENT_CACHE_SIZE : Size of client's cache.
SERVER_CACHE_SIZE : Size of server's cache.
BLOCK_SIZE : Size of individual block
DISK_ACC_TIME : Server disk access time.
CLIENT_CACHE_ACC_TIME: Client cache access time.
SERVER_CACHE_ACC_TIME : Server cache access time.
PROCESSIN_TIME : Processing time for data.
NETWORK_LAT : Network latency time for data along the network.
MANAGER_NAME : Name of manager
LOWER_LIMIT : File size lower limit
UPPER_LIMIT : File size upper limit
STRIDED : Random seed
TYPE : Type of test case BEST1, BEST2, DISK, RANDOM
TOTAL_FILE_READ : Total file reads
MAX_APPLICATION : Number of applications running
DISCARDED_CACHE_SIZE : Discarded cache size
PREFETCH_COUNT : Pre-fetch count for OBA

Pre-fetching algorithm integration

Each client keeps track of the number of blocks from a file they are accessing. The manager in our hint-based cooperative cache forwards this information as part of the hint to the requesting clients. The hint-based cooperative caching is the most decentralized cooperative caching schema. Each of a client manages its own cache using local information and contacts the manager only if a block is not present in the local cache or local hints. In our modified hint-based algorithm, each of the client manages its pre-fetching locally since each client has only the local information of the file access pattern.

OBA pre-fetching algorithm is most widely used pre-fetching scheme. The algorithm has fixed degree of pre-fetch. When a block is requested we also bring the next three blocks in the cache before it is requested. For integration of the IS_PPM pre-fetching scheme, our modified client constructs the graph of access pattern for the file as discussed in the above section and store them locally for future use.

For our AAP algorithm, each client initiates the pre-fetching of the data structure to keep track of P and G values. All applications running on the client share the same data structure for P and G. When client initiates block lookup for a first block, it brings p set of blocks in the sequence and asynchronous pre-fetching is activated. We initialize the value of P to three and G to zero. We also need to distinguish between lookup block and pre-fetched block. For this purpose each block object holds the pre-fetched flag. Each application maintains its current pre-fetch set. This set is used to increment and decrement P and G values as discussed above.

V.IMPLEMENTATION AND SIMULATION METHODOLOGY

Package structure

We have used the discrete event simulation methodology for our simulation environment. The code structure is housed under the simulation package as explained below.

- Client – This package contains an abstract class Client, which is the base class and provides all necessary methods for client's housekeeping. Hint-Based, OBA, IS_PPM and AAP clients are derived classes and simulate the respective algorithms as explained in the Section *. Each client contains LRU cache object as described below and also maintains global and local hit counts and total block reads.
  - Disk – Package houses different objects to simulate disks, files and blocks. Disk class generates the required files and blocks based on pseudo-random generator. The size of files is driven by UPPER_LIMT and LOWER_LIMIT parameter of the configuration file. Block objects contain master, pre-fetched and accessed flags. Whenever client accesses the block its access block is set to true. Access and pre-fetched flag are reset, while forwarding the block to another client as part of the best-guess replacement policy.
  - Server – Package contains the server object implementation. The server objects use the same LRU cache implementations as the clients.
  - Cache – This package houses the LRU cache implementations. This custom implementation allows us to vary the size of cache.
- Simulator – This is the main package that contains the Start, Config and Graphgenerator objects. The Config class reads the necessary simulation parameters from the configuration file. After reading configuration parameters, an object of the Start class initiates the simulation. The start class contains the implementation for generating discrete events. Each client contains an application that performs the read request as instructed by the Start object on the respective client. After completion of all read requests, the Graphgenerator object collects the necessary data and generates the graphs.

Simulation methodology

We have used synthesized workloads instead of using real traces. This has allowed us to consider those extreme cases that otherwise would not be possible in the case of traces. The cache and the file sizes together impact the performance of the pre-fetching algorithms. Each of the modified clients runs multiple applications requesting a different file for reading. We have used pseudo-random number generator from the PJ library [cite] for selecting a file from given N number of files. For given N, it generates random number with 1/N probability.

For our test cases, two types of streams are considered namely single stream and multiple streams. In the single stream, at a given time only one application is requesting access to the file on same client. On the other hand, in multiple streams more than one application is requesting the files. We have to also consider different lengths of files for our simulation. Pre-fetching is affected by number of streams requesting files and length of the sequences. Hence we have considered four different test cases – multiple streams with long sequences, multiple streams with short sequences, single stream with long sequences and single stream with short sequences. We have also run three ideal setups for evaluating best-case and worst-case conditions. These setups provide base conditions for other test cases.

Experiment and Result Analysis

All the experiments were performed on intel Xeon X3440 processor capable of running 12 threads with 12GB internal memory. This section provides analysis of an experiment conducted as described in the proposal section.

Evaluation Metrics

In order to evaluate the performance of the proposed algorithm, we have used following evaluation metrics.

Average block access time: This is the ratio of time taken by a client to locate the file block and fetch it into the cache against total number of clients.

Global cache hit ratio: This is the ratio of the number of blocks found in other client's cache against total number clients. Average local cache miss ratio: This is the ratio of the number of blocks found in a client's local cache to total clients. Disk access time: This is the ratio of the total number of disk reads against the total number of clients. Pre-fetch wastage: This is the percentage of pre-fetched wastage against total number of clients. Pre-fetched wastage is (un

accessed blocks evicted / total blocks evicted).

Base Cases

All blocks available in local cache

This base case simulates ideal setup where all blocks are available in the client's local cache. We have to first fill the client cache with set of files and later tell the clients to read the same set of files. Figure 5 shows an average access time comparison between four algorithms for this base case. All the four algorithm's average access time is same as that of the client cache access time. This case provides optimal average time for accessing the file blocks.

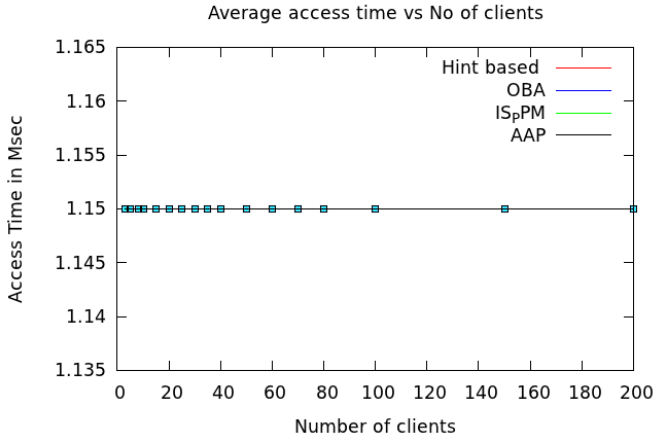

Fig 5: Average access time VS number of clients

All blocks available in remote client cache

This base case simulates setup where all requested blocks are available in peer client's cache. We first fill the client cache with set of files and later tell the clients to read the set of files that are not present in their own cache. This case provides base condition for when the blocks are located in the global cache. Figure 6 shows an average access time comparison between the four algorithms. All the pre-fetching algorithms outperform the hint-based algorithm. Access time for pre-fetching algorithms is less as compared to the hint-based since time taken to fetch N blocks one at a time is more than when they are fetched simultaneously as is the case in the pre-fetching algorithms.

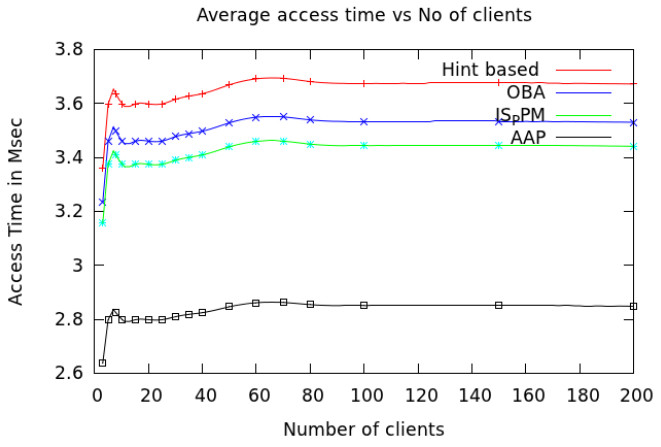

Fig 6: Average access time VS number of clients

All blocks are read from the disk

This base case simulates setup where all blocks are read from the disk. We tell each client to read unique set of files. This case provides the worst-case scenario for all four algorithms. Figure 7 shows an average access time comparison between the four algorithms. As observed, the disk access time for the hint-based algorithm is more than the pre-fetching algorithms. Modern disks store the file blocks in contiguous memory locations [reference]. This provides an added advantage while accessing contiguous blocks. Because of this time for accessing N blocks in the single I/O request is less than that of N number of I/O requests. Due to an adaptive degree of pre-fetch the AAP algorithm outperforms the other three algorithms with respect to average access time.

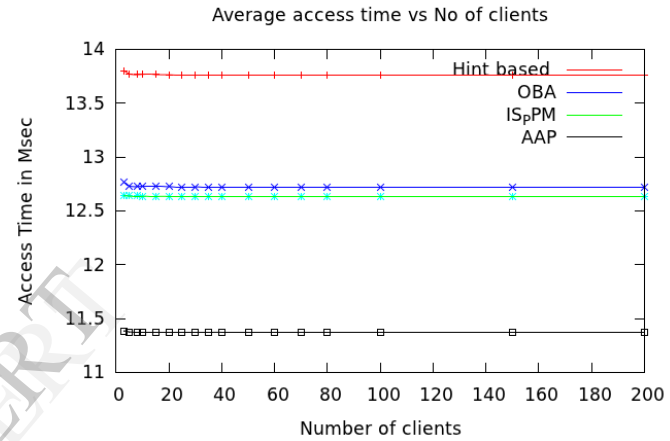

Fig 7: Average access time VS number of clients

Test Cases

In each of the following test case we have run four clients with the same number of blocks and the same set of files. Configuration parameters used for each test case are listed in the appendix A. The collected data for each test case is listed in a tabular format in the appendix B. We have used the same time constants for all the test cases in Table 1. All the timings are in microseconds and the block size is in kilobytes.

Multiple streams with long sequences

In this test case we have used maximum of three applications on each client. We have used files with long sequences with an average of 51 blocks per file. Figures 8, 9 and 10 show the performance comparison between the four algorithms. In Figure 9, we observe that running multiple applications that are requesting files with long sequence impacts the cache wastage. After 40 clients the cache wastage is down to zero. Figure 8 plots the comparison analysis of the average access time for all the four algorithms. Figure 10 plots local hit and miss, global hits and disk read ratios. The local miss and hit ratio plots are proportional to each other. As observed, all the algorithms produce same global hits and disk reads. This is because all pre-fetching algorithms are simulated on top of the hint-based model. In Figure 10 (c), the pre-fetching algorithms show

fewer number of disk reads as compared to the hint-based algorithm.

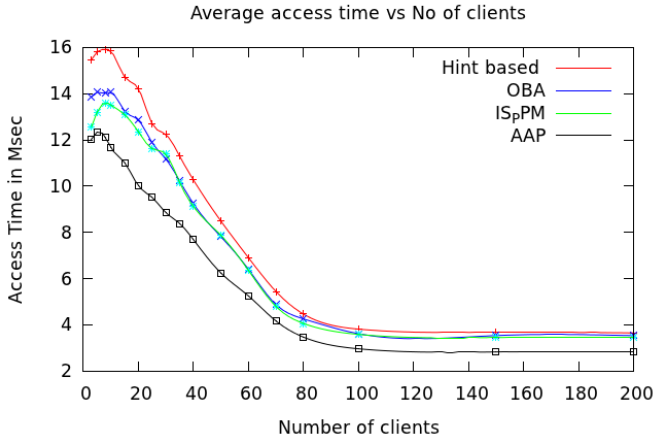

Fig 8: Average access time VS Number of clients

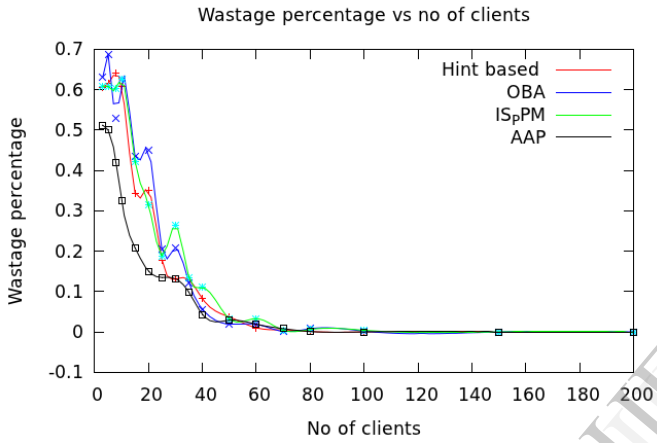

Fig 9: Perecentage of wastage VS number of clients

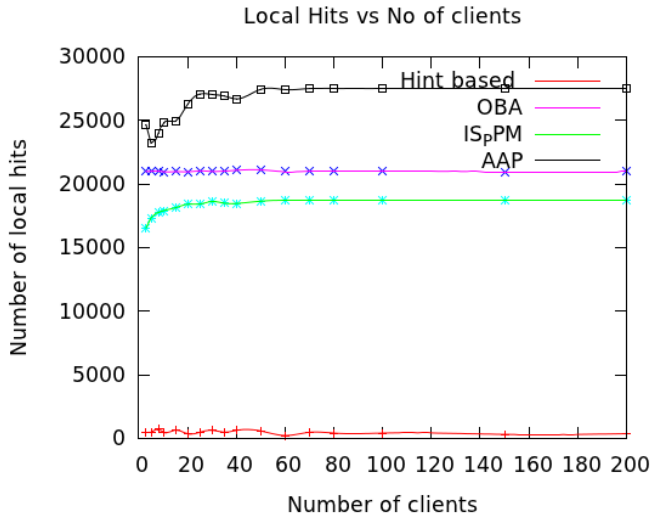

Fig10: a) Local hit ratio

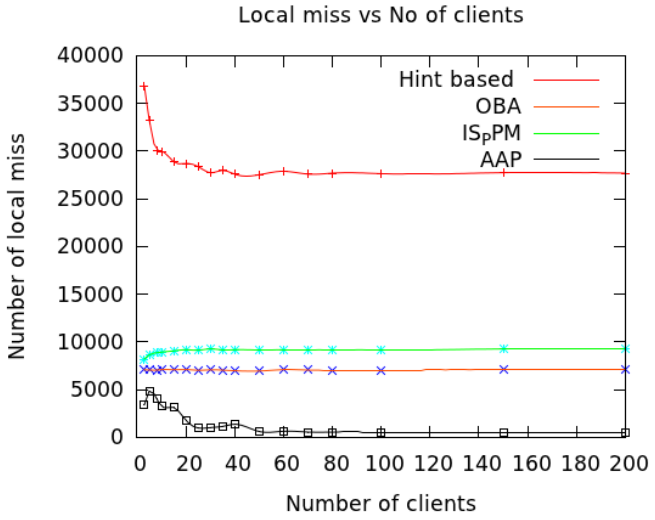

Fig10: b) Local miss ratio

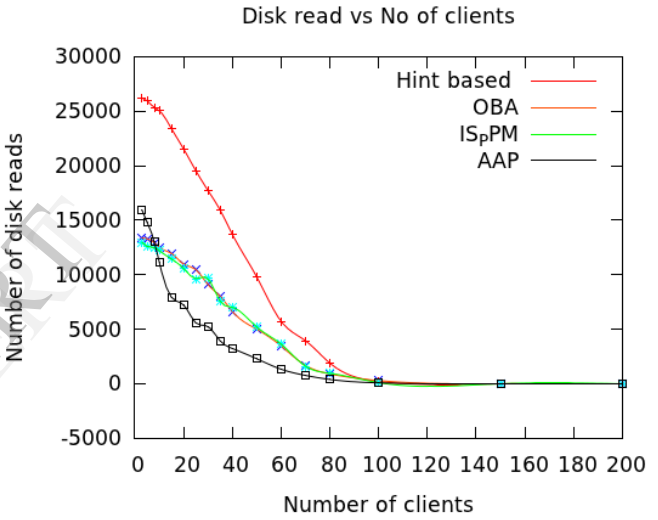

Fig10:c) Disk read ratio

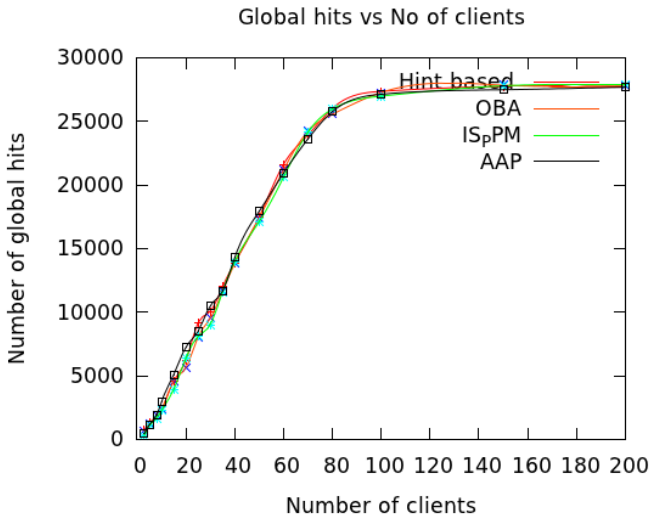

Fig10 :d) Global hit ratio

Multiple streams with short sequences

In this test case we have used maximum of three applications on each client. We have used files with short sequences with an

average of 15 blocks per file. Figures 11, 12 and 13 show the performance comparison between the four algorithms. Figure 11 plots the comparison analysis of the average access time for all the four algorithms. Figure 13 plots local hit and miss, global hits and disk read ratio. As observed, due to short sequences the average access time is reduced significantly as compared to long sequences in the case of AAP. This is because of AAP algorithm adapts to high pre-fetch degree for short sequences.

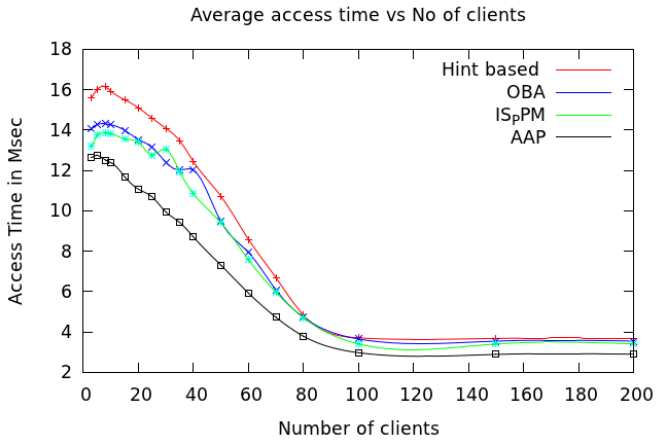

Fig 11: Average access time VS Number of clients

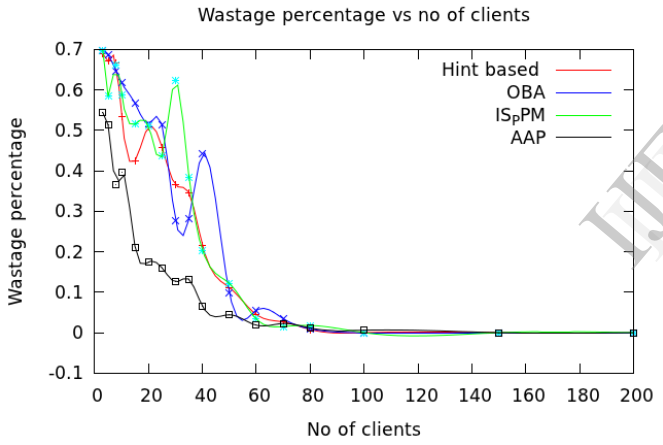

Fig 12: Perecentage of wastage VS number of clients

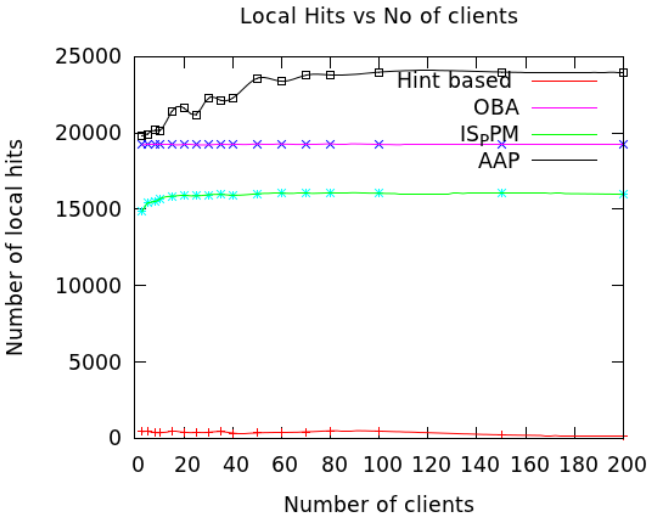

Fig 13: a) Local hit ratio

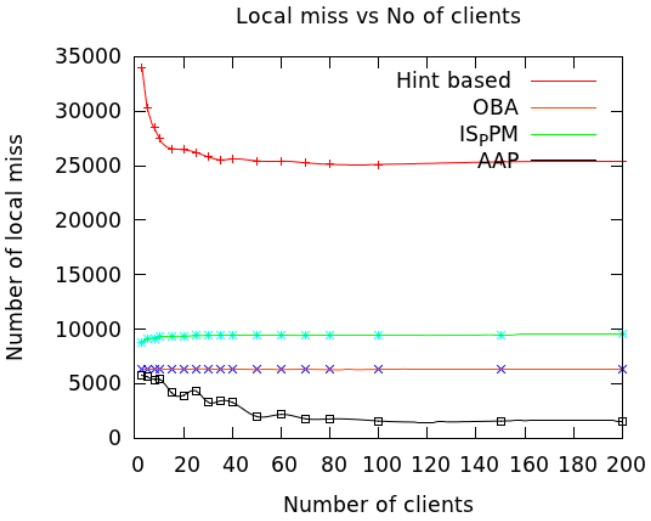

Fig 13: b) Local miss ratio

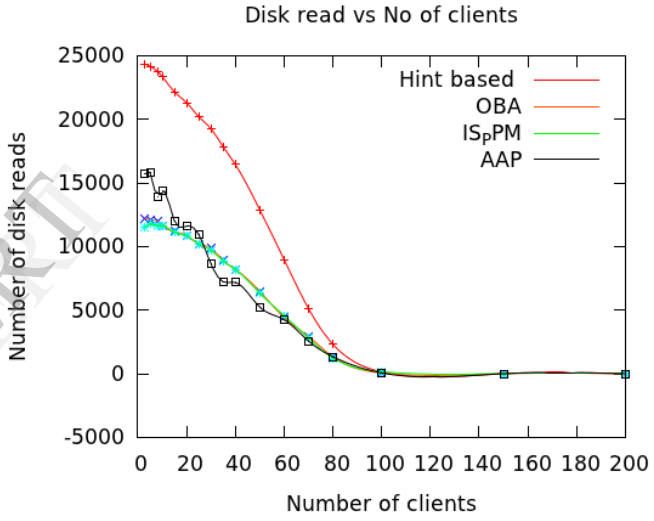

Fig 13: c) Disk read ratio

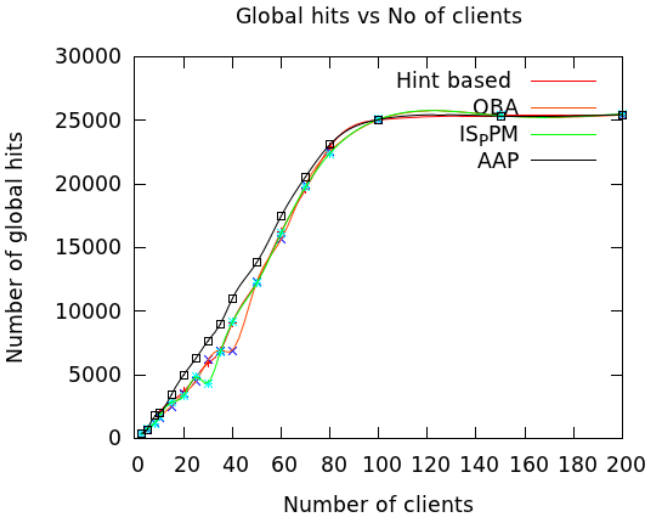

Fig 13:d) Global hit ratio

Single streams with long sequences

In this test case we have run a single application on each client and used the files with long sequences with an average of 16 blocks per file. Figures 14, 15 and 16 show the performance comparison between the four algorithms under given simulation parameters. Figure 14 plots comparison analysis of the average access time for all the four algorithms. Figure 16 plots local hit and miss, global hits and disk read ratio. Figure 15 shows the ratio of percentage of cache wastage to the total number of clients. We observe that the cache wastage is less as compared to the previous test cases.

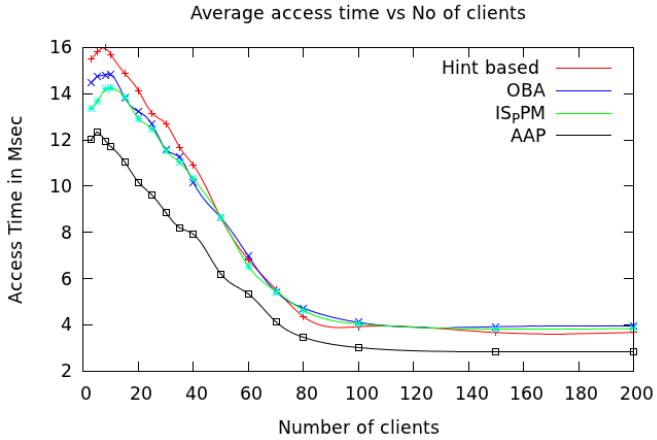

Fig 14: Average access time VS Number of clients

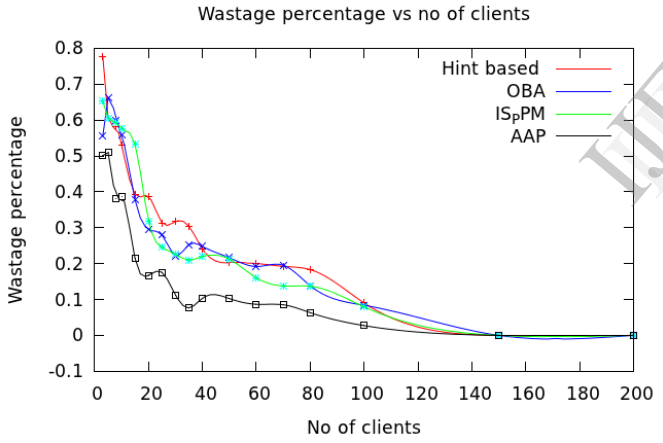

Fig 15: Perecentage of wastage VS number of clients

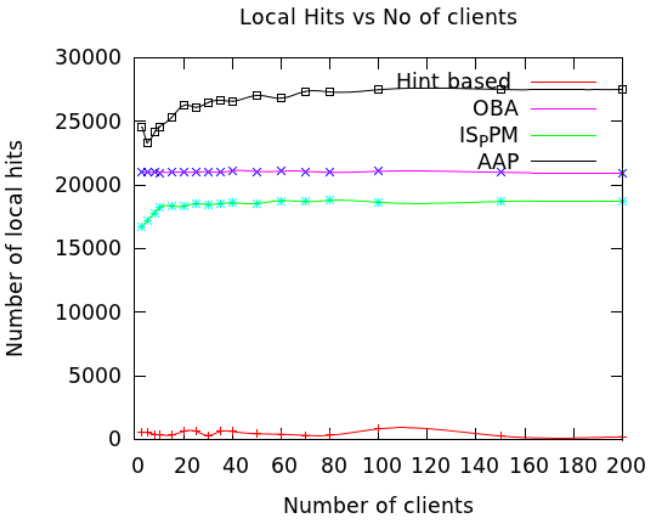

Fig 16: a) Local hit ratio

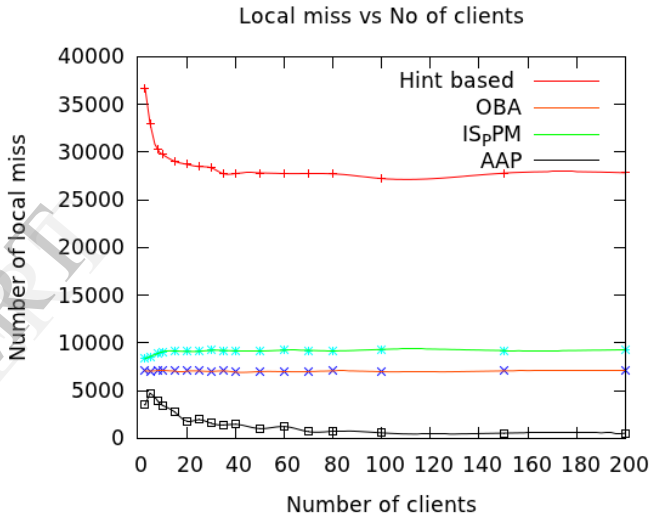

Fig 16: b) Local miss ratio

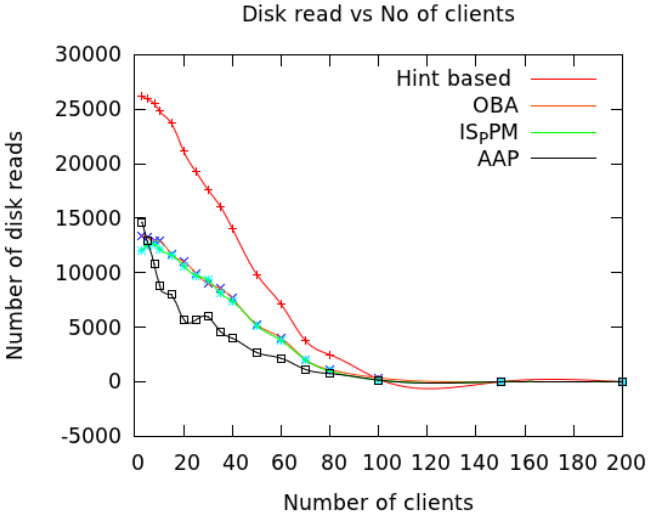

Fig 16: c) Disk read

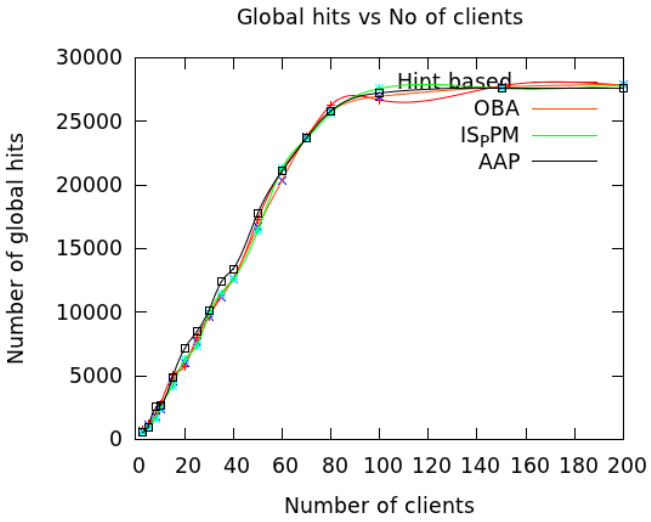

Fig 16: d) Global hit ratio

Single streams with short sequences

In this test case we have again run a single application on each client. But now we have used files with short sequences with an average of 15 blocks per file. Figures 17, 18 and 19 show the performance comparison between the four algorithms. Figure 19 plots local hit and miss, global hits and disk read ratio. Figure 17 plots comparison analysis of the average access time for all the four algorithms. We observe that the average access time and the cache wastage in the pre-fetching algorithm are less compared to the previous test cases. This is because, this test case has only one application running on the client that requests short sequences.

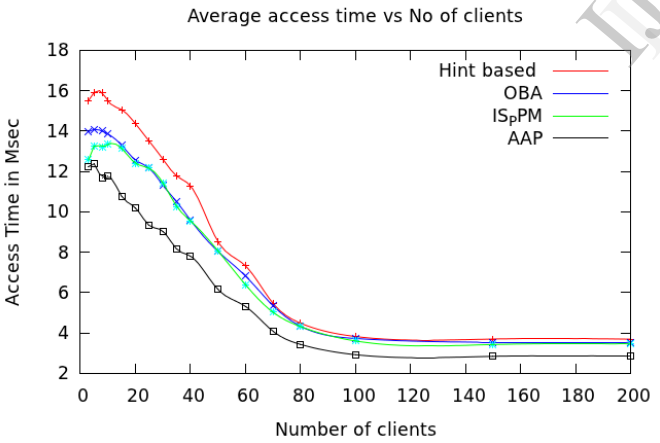

Fig17: Average access time VS Number of clients

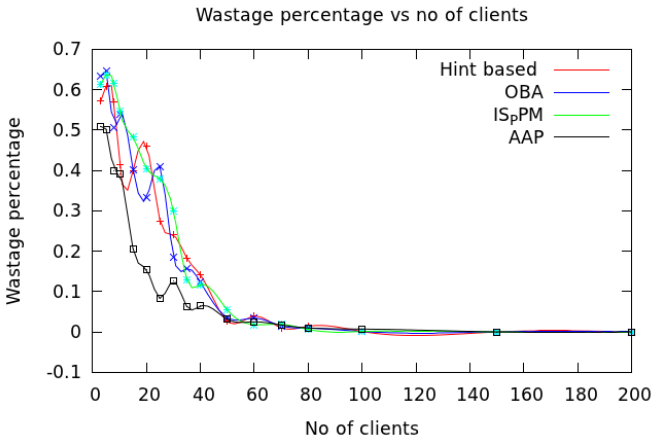

Fig18: Perecentage of wastage VS number of clients

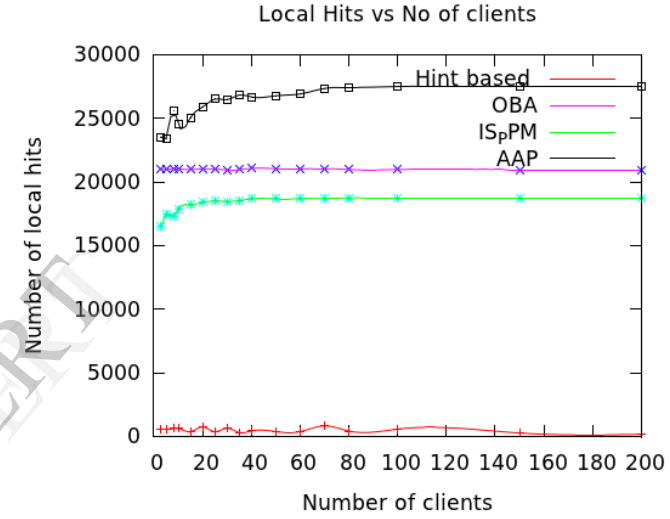

Fig 19: a) Local hit ratio

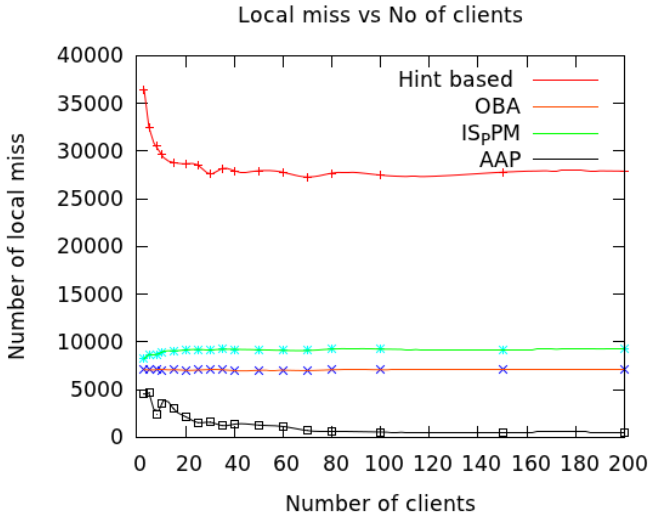

Fig 19: b) Local miss ratio

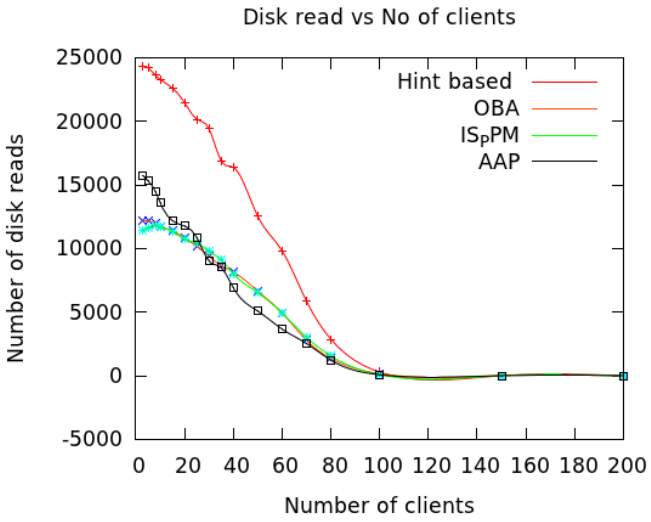

Fig 19: c) Disk read

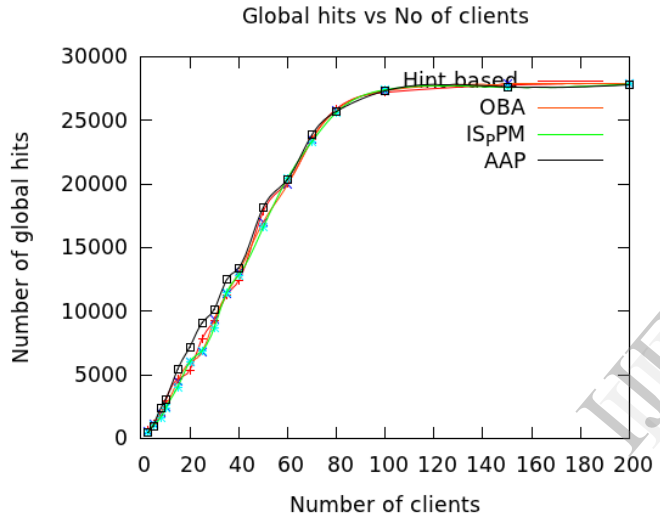

Fig 19: d) Global hit ratio

V .CONCLUSIONS

This project is aimed at reducing the problem of cache wastage and cache pollution and integration of pre-fetching scheme in hint-based cooperative caching. The simulation results above show that AAP algorithm outperforms the other three algorithms.

- Average access time: Figures 8, 11, 14 and 17 show that the average access time taken to access the block is less than that of the other three algorithms. As from the graphs, average access time is less in the test cases where single application is running.
- Cache wastage: Figures 9, 12, 14 and 18 plot cache wastage for the four test cases. We have observed that due to an improved LRU policy of AAP algorithm cache wastage has reduced significantly in all the four test cases.
- Disk reads: Figures 10(c), 13(c), 15(c) and 19(c) plot the total disk reads against total number of clients. For 3 to 15 clients the total available cache is small for large pre-fetching of blocks. The AAP algorithm thus maintains small pre-fetching degree for reducing cache wastage. Because of this AAP algorithm has more disk access than OBA and IS_PPM for small number of clients. As the total memory is increased the AAP algorithm thus outperforms OBA , IS_PPM and the hint-based algorithms.
- Local hits/miss ratio: Figures 10(a), (b), 13 (a), (b), 15 (a), (b) and 19(a), (b) plot the local hit and local miss ratios. Due to asynchronous nature of AAP the local misses are significantly reduced.

ACKNOWLEDGMENT

I would like to thank Professor Hans Peter Bischof for his valuable guidance and support throughout implementation of my master's capstone project. I would also like to thank Professor James Heliotis for his valuable inputs on project document tation.

References

[1] E. Anderson, C. Hoover, and Xiaozhou Li. New algorithms for _le system cooperative caching. In Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium on, pages 437 {440,aug. 2010.
[2] T. Cortes and J. Labarta. Linear aggressive prefetching: a way to increase the per-formance of cooperative caches. In Parallel Processing, 1999. 13th International and 10th Symposium on Parallel and Distributed Processing, 1999. 1999 IPPS/SPDP.Proceedings, pages 46{54, April 1999.
[3] Fredrik Dahlgren, Michel Dubois, and Per Stenstrom. Fixed and adaptive sequential prefetching in shared memory multiprocessors. In Parallel Processing, 1993. ICPP 1993. International Conference on, volume 1, pages 56{63, aug 1993.
[4] Binny S. Gill and Luis Angel D. Bathen. Optimal multistream sequential prefetching in a shared cache. Trans. Storage, 3(3), October 2007.
[5] P. Kristian K. M. Curewitz and J. S. Vitter. Practical prefetching via data compres-
sion. In SIGMOD Management of Data, pages 257{266, 1993.