11-19-2007

AMP: An Affinity-based Metadata Prefetching Scheme in Large-Scale Distributed Storage Systems

Lin Li
University of Nebraska - Lincoln, lili@cse.unl.edu

Xuemin Li
Chongqing University, Chongqing, China

Hong Jiang
University of Nebraska - Lincoln, jiang@cse.unl.edu

Yifeng Zhu
University of Maine Orono, ME

University of Nebraska–Lincoln, Computer Science and Engineering
Technical Report TR-UNL-CSE-2007-0024
Issued Nov. 19, 2007

AMP: An Affinity-based Metadata Prefetching Scheme in Large-Scale Distributed Storage Systems

Lin Lin, Xuemin Li^, Hong Jiang, Yifeng Zhu*
Computer Science and Engineering
University of Nebraska-Lincoln, Lincoln, Nebraska 68588
Email: {lilin, jiang}@cse.unl.edu
^College of Computer Science
Chongqing University, Chongqing, China 400030
Email: lixuemin@cqu.edu.cn
*Electrical and Computer Engineering
University of Maine Orono, ME 04469-5708
Email: zhu@eece.maine.edu

Abstract Prefetching is an effective technique for improving file access performance, which can reduce access latency for I/O systems. In distributed storage system, prefetching for metadata files is critical for the overall system performance. In this paper, an Affinity-based Metadata Prefetching (APM) scheme is proposed for metadata servers in large-scale distributed storage systems to provide aggressive metadata prefetching. Through mining useful information about metadata assesses from past history, AMP can discover metadata file affinities accurately and intelligently for prefetching. Compared with LRU and some of the latest file prefetching algorithms such as NEXUS and C-miner, trace-driven simulations show that AMP can improve the hit rates by up to 12%, 4.5% and 4%, respectively, while reduce the average response time by up to 60%, 12% and 8%, respectively.

Index terms: Prefetch, metadata, storage, data mining

1. Introduction and Motivations

High-performance computer system designers have long sought to improve the performance of file systems, which have proved critical to the overall performance of an exceedingly broad class of applications. The scientific and high-performance computing communities in particular have driven advances in the performance and scalability of distributed storage systems. Since all I/O requests can be classified into two categories, user data requests and metadata requests, the scalability of accessing both data and metadata has to be carefully maintained to avoid any potential performance bottleneck along all data paths. A novel decoupled storage architecture diverting actual file data flows away from metadata traffic has emerged to be an effective approach to alleviating the I/O bottleneck in modern storage systems [1]-[4], as shown in Figure 1. In such a system a client will consult a metadata server (MDS) cluster, which is responsible for maintaining the file system namespace, to receive permission to open a file and information specifying the location of its contents. Subsequent reading or writing takes place independently of the MDS cluster by communicating directly with one or more storage devices [5][6]. Previous studies on this new storage architecture mainly focus on optimizing the scalability and efficiency of file data accesses by using RAID-styled striping [7], [8], caching [9], scheduling [10] and networking schemes [11].

However, while the scalability of metadata operations is also very critical, it tends to be ignored or under estimated. Metadata not only provides file attributes and data block addresses, but also synchronizes concurrent updates, enforces access control, supports recovering and maintains consistency between user data and file metadata. A study on the file system traces collected in different environments over a course of several months show that metadata operations may make up over 50% of all file system operations [13], making the performance of the MDS cluster of critical importance. Furthermore, while the overall capacity of the storage server cluster can easily scale by increasing the number of (relatively independently operating) devices, metadata exhibits a higher degree of interdependence, making the design of a scalable system much more challenging.

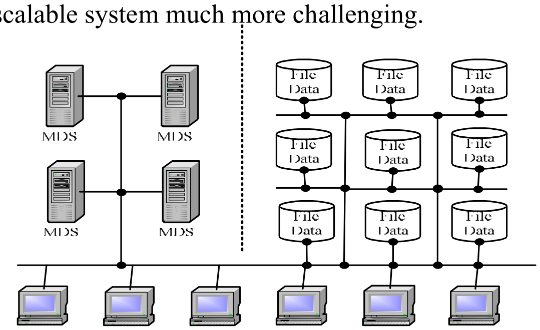


Figure 1 System architecture

Existing caching and prefetching schemes designed for and applied on actual file data typically ignore metadata characteristics [14]. The most important characteristic of metadata is its much smaller size relative to actual file contents. Conventional data prefetching algorithms are usually very conservative and only prefetch one or two files upon each cache miss. They are not efficient for metadata prefetching. Because of relatively small size of metadata, the miss-prefetching penalty for metadata on both the disk side and the memory cache side is likely much less than the penalty for file data miss-prefetching [14]. Hence, an aggressive prefetching algorithm is desirable for metadata in order to handle large-volume of metadata traffic.

This paper proposes an affinity-based metadata prefetching (AMP) scheme that applies data mining techniques to discover and identify the affinities existing among metadata accesses from past history and then uses these affinities as hints to judiciously perform aggressive metadata prefetching. The main technical contribution of this paper includes.

1. It develops an aggressive but efficient affinity-based metadata prefetching algorithm based on data mining techniques. The experimental results show that we can prefetch up to 6 metadata files at one time.
2. AMP explores the impacts of different parameters (such as prefetching group size, server-oriented vs. client-oriented prefetching, group header size) to optimize the tradeoff between the efficiency of metadata prefetching, and the memory and network overhead.
3. It compares AMP with some of the state-of-the-art prefetching schemes, including the NEXUS metadata prefetching algorithm [31] and the block-correlation-discovery C-Miners algorithm [30], qualitatively and quantitatively. Comparison results show that AMP consistently outperforms both NEXUS and C-Miners.

The rest of the paper is organized as follows. Section 2 outlines existing relevant algorithms to provide a background for AMP. Section 3 describes the proposed algorithm and discusses its design issues. The simulation methodology and the performance evaluations are presented in Section 4. Section 5 concludes the paper.

2. Related Work

In this section, we briefly discuss some representative work that is closely related to this paper. Data prefetching has been studied extensively in databases, file systems and I/O-intensive applications. Most of previous prefetching work either relies on applications to pass hints [15-19] or is based on simple heuristics such as sequential accesses. Ref. [20] is an example of prefetching in disk caches. I/O prefetching for out-of-core applications including compiler-assisted prefetching is proposed in [21, 22] and prefetching through speculative execution is introduced in [23].

In the spectrum of sophisticated prefetching schemes, research has been conducted for semantic distance-based file prefetching for mobile or networked file servers. The SEER project from UCLA [24, 25] groups related files into clusters by keeping track of semantic distances between files and downloading as many complete clusters as possible onto the mobile station. Kroeger extends the probability graph to a tree with each node representing the sequence of consecutive file accesses from the root to the node [26]. Lei and Duchamp also use a similar structure by building a probability tree [27, 28].

There are also some studies on metadata prefetching. Nexus [31] is a weighted-graph-based prefetching technique, built on successor relationship, to gain performance benefit from prefetching specifically for clustered metadata servers.

Data mining methods have been mostly used to discover patterns in sales, finance or bio-informatics databases [29]. A few studies have applied them in storage systems. For example, Li et al. [30] proposed C-Miner using data mining techniques to find block correlations on storage server to direct prefetching.

STEP [32] proposed a sequentiality and thrashing detection-based prefetching scheme to aggressively prefetch disk data based on cost-benefit analysis for two typical storage access patterns: sequential access patterns and disk thrashing patterns.

3. Affinity-based Metadata Prefetching Scheme

In this section, we will introduce our new data mining based metadata prefetching algorithm AMP. AMP explores deep affinities from metadata files and it involves two steps: (1) It first analyzes past metadata access history and extracts connotative relevancy for each file metadata and (2) It then utilizes the small size characteristic of file metadata and aggressively prefetches multiple metadata simultaneously. Since file metadata typically are much smaller than actual file contents, the penalty for metadata miss-prefetching would be relatively smaller compared to data miss-prefetching.

A. Metadata Affinities

Metadata affinities widely exist in storage systems. The metadata of two or more files are affined if they are "linked" together either spatially or temporally. For example,*/usr* always has a strong spatial affinity with */usr/bin*, */usr/bin/ls* and */usr/bin/ps*. If we can find out the strong affinities between these metadata, we could prefetch all these metadata files into cache simultaneously. This can potentially significantly reduce the response time, especially in distributed storage systems, where we need to obtain such metadata files from remote MDS.

B. Affinity Identification

AMP uses the recent metadata access history and applies data mining techniques to discover metadata affinities. For example, it can use one week's trace to train the algorithm to extract the affinities, and then use this

```
F ← NULL //F is a forest
for each item m_i of M do
  if (m_i does not exist in F)
    add m_i to F
end for
for i←1 to n-1
  G_i =m_i m_(i+1)…m_(i+w-1) // history window size w
  G_i ←filter ( G_i ) //filter: fix first two items in G_i and remove same items in G_i
  group S_i ←m_i m_(i+1)+ subset( G_i =m_(i+2)…m_k ) //fix first two items of G_i , concatenate with the rest items in G_i
    for each S_i do
      search m_i in F
      if (children of node m_i don't contain node m_i m_(i+1))
        add node m_i m_(i+1) under node m_i
      else
        frequency of m_i m_(i+1) + 1
      j←3
      while j<=length ( S_i )
        find m_i m_i+1 … m_j-1
          if (children of m_i m_i+1 … m_j-1 do not contain m_i m_i+1 … m_j)
            add node m_i m_i+1 … m_j under m_i m_i+1 … m_j-1
          else
            frequency of m_i m_i+1 … m_j +1
          j++
      end while
    end for
end for
MaxGroups(all trees in F) //for each tree, compare frequency of every node under level 2 and find out the node who has the biggest frequency
```

affinity information for metadata prefetching during the next week. A prefetching window with a fixed capacity is adopted in AMP. The prefetching window will move when a new request arrives. In the prefetching window, we fix the first two items as a header and concatenate the rest items with the header to form a sub–sequence. The pseudo-code of our algorithm above is provided to describe how AMP works.

We use an example to illustrate the basic idea of our algorithm. Suppose that the history window size is six and a request sequence is given as follows

D= {ABCADEFABE}

As illustrated in Figure 2 the procedure divides the sequence into fixed-length segments by moving the history window sequentially.

{ABCADE} FABE

A {BCADEF} ABE

AB {CADEFA} BE

ABC {ADEFAB} E

ABCA {DEFABE}

Figure 2 History window movements

For each segments, the first two file metadata are considered as the prefix group and the set of the latter four file metadata excluding those in the prefix are the affix group. For example, in the segment {*ABCADE*}, the affix {*CDE*} does not include *A* since *A* is in the prefix. The basic idea is that a prefix group gives positive support for prefetching to all elements in the affix. For example, for the segment {*AB:CDE*} if *A* and *B* are accessed, {CDE} are likely to be accessed again in the future. The following shows the details of all prefix and affix groups for all segments obtained by moving the window sequentially along the access sequence.

{ABCADE} ⟶ {AB: CDE}

{BCADEF} ⟶ {BC: ADEF}

{CADEFA} ⟶ {CA: DEF}

{ADEFAB} ⟶ {AD: EFB}

{ABCADE} ⟶ {AB: CDE}

{DEFABE} ⟶ {DE: FAB}

{EFABE} ⟶ {EF:AB}

{ABE} ⟶ {AB:E}

{BE} ⟶ {BE}

Figure 3 Group information

An access forest will be built with all accessed file metadata in the near past as roots, as shown in Figure 4.

A B C D E F

Figure 4 tree root nodes

Then, each root node is extended into a weighted access tree by adding all prefix-affix pairs. For example, for the prefix-affix pair {*AB:CDE*}, AB will be added to the tree as level one node. Then *ABC*, *ABD*, *ABE* will be added to the tree as level 2 nodes. After that, *ABCD*, *ABCE*, *ABDE* will be added to the tree. Then, the last one *ABCDE* would be added to the tree, as shown in figure 5.

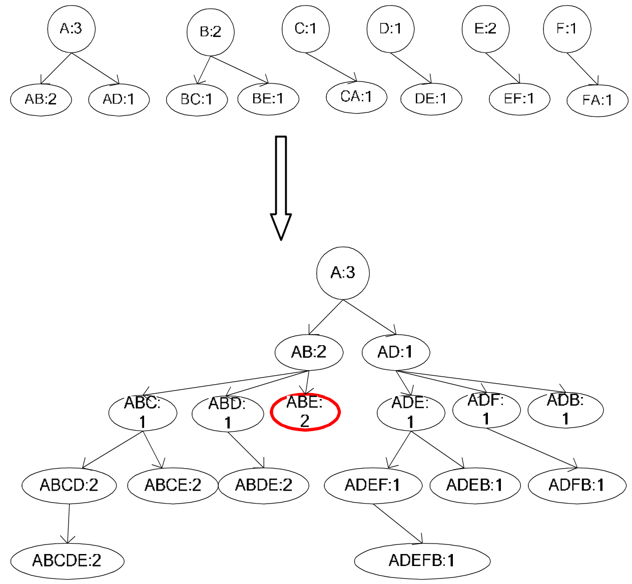


Figure 5 Training results

From the training result of *A*, as shown in Figure 5, we can find that the frequency of node *ABE* is 2, which is larger than the weights of the other path rooted from *A*. This indicates that *ABE* has a strong affinity. When item *A* or *AB* appears, E is most likely to be accessed in the very near future. This obtained affinity is what we need for prefetching.

Many prefeching algorithms use only the currently accessed object to predict the objects that are likely to be accessed in the near future. Such approaches are believed to be neither accurate nor adequate. Accordingly, AMP chooses to use multiple objects, instead of the currently accessed one, to perform predictions. For example, given a group *ABCDEF*, if *A* is already in the cache, and a cache miss happens on *B*, the prefetching affinity should be $AB \rightarrow CDEF$, instead of $B \rightarrow CDEF$. Using *AB* simultaneously provides a better prefetching accuracy. This is base on the fact that

$$P(Group \mid X_1) < P(Group \mid X_1 X_2)$$

AMP has the following major advantages. Firstly, the most significant difference between AMP and other probability based approaches is that AMP is not limited to predicting the most immediate successor. AMP aims to provide a deeper insight into the future and aims to predict a group of metadata that are likely to be accessed for aggressive prefetching.

Secondly, AMP provides more accurate predictions. Nexus constructs a graph for all items and selects those items with largest weight for prefetching. The relations between file metadata are relatively simple and straight. In addition, the affinity identified by Nexus is sometimes inaccurate under some circumstance. Typical prefetching rules in Nexus are similar to this: $A \rightarrow CD$ (Upon a miss on *A*, Nexus prefetches *C* and *D*). AMP explores the affinity with longer prefix, such as *AB->CD* in which *A* is in cache and a miss happens on *B*. AMP uses both *A* and *B* to determine the prefetching of *CD*. This design with longer prefix helps to reduce mis-predictions and also improve the capability of predicting further into the future. In addition, our experiments show that when the prefix length increases to 3 or 4, the prefetching accuracy almost has no significant improvement, while the algorithm complexity increases exponentially.

Thirdly, AMP is more aggressive than Nexus by taking advantage of the fact that file metadata typically are small in size. In real-trace experiments, we have found that AMP can prefetch up to 6 file metadata during a cache miss, while Nexus only perfetches 2 file metadata.

Similarly to other algorithms, AMP can also perform affinity discovery in an on-line fashion without system-level intervention. For example, AMP can train each day trace at midnight and use the training results for the second day's prefetching. The new training results are accumulated into the database while old results in the database are either replaced or aged over the time. In this aspect, AMP differs from C-Miner that only uses recent traces for training and training results are not accumulated.

Another important difference between AMP and C-Miner is that AMP has less overhead. AMP places more focus on affinity and less on strict access orders. Fox example, AMP treats the following prefix-affix pair exactly the same in identifying affinity: $A \rightarrow BCDE$ and $A \rightarrow DEBC$, while C-Miner considers them to be different for prefetching. Accordingly C-Miner identifies few affinity sequences, thus less accurate.

C. Design issues

C.1 Prefetch group size

The size of file metadata is typically uniform and much smaller than the size of file contents in most file systems. With a relatively small size, the penalty for miss-prefetching on both the disk side and the memory cache side is likely much less than that for file data, allowing the opportunity for exploring and adopting more aggressive prefetching algorithms. We study the impact of the impact of prefetch group size from 3 to 9, as shown in Figure 6. It is interesting to observe that the hit rate remains almost unchanged when the group size increases from 7 to 9. Thus, in this paper, we choose to use 8 as the group size. This means that when the size of prefix group is two, we can prefetch up to 6 items for one cache miss. .

C.2 Header size

In this part, we will analysis the hit rate and the prefetch header size. This header size is also referred to as the prefix N is the N-gram scheme. An N-gram is a sub-sequence of n items from a given sequence. N-grams are used in various areas of statistical natural language processing and genetic sequence analysis. When we fix the first item of the group, we call it two-gram, fix the first two items of the group, we call it three-gram and so on. Instinctively, when the header size increases, the prefetching accuracy is expected to increase, while the algorithm complexity increases exponentially. Figure 7 shows the prefetch performance of 2-gram, 3-gram, 4-gram and 5-gram. Compared with 3-gram, 4-gram or 5-gram cannot provide no improvement. Thus, in this paper, 3-gram is chosen in AMP.

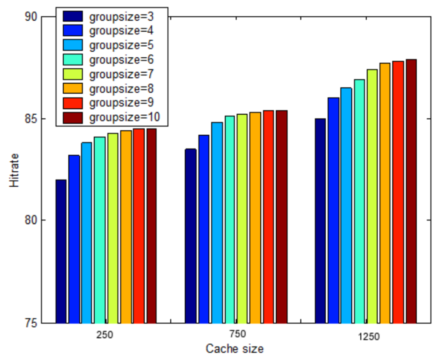


Figure 6 Group size comparison

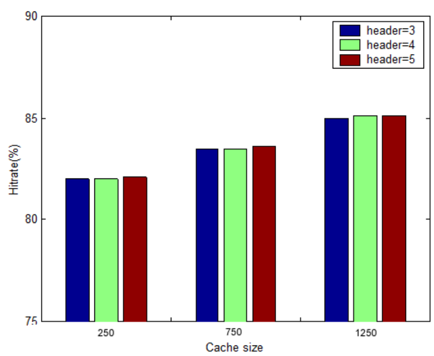


Figure 7 N-Gram header size

C.3 Server-oriented grouping vs. client-oriented grouping

There are two different approaches to affinity discovery: 1) obtain affinities for all requests received by a particular metadata server; or 2) obtain affinities for requests sent separately from individual clients. In this paper, we refer to the former as server-oriented access grouping, and the latter as client-oriented access grouping [31]. Our experimental results prove that the client-oriented scheme always out-performs the server-oriented scheme. Thus, the client-oriented grouping is chosen in our design.

4. Performance Evaluation

We use trace-driven simulations to evaluation our design based on several large traces collected in real systems. We have developed a metadata management simulator that incorporates widely used DiskSim simulator [33].

A. workloads

To the best of our knowledge, there are no publicly available file system traces that have been collected from a large scale cluster with thousands of nodes. We conduct our simulations on two public traces: the HP traces [34] and Harvard SOS Traces [35, 36]. HP traces are 10-day long file system traces collected on a time-sharing server with a total of 500GB storage capacity and 236 users. To emulate the I/O behaviors of such a large system and facilitate a meaningful simulation, we artificially scale up the workloads from 200 clients to about 8000 clients by merging multiple trace files into one, thus increasing the access density while maintaining the time order of access sequences. Harvard SOS traces are collected from the some departments and main campus general-purpose servers with a total of 160 GB. We use the one collected from the main campus general-purpose servers for our simulation.

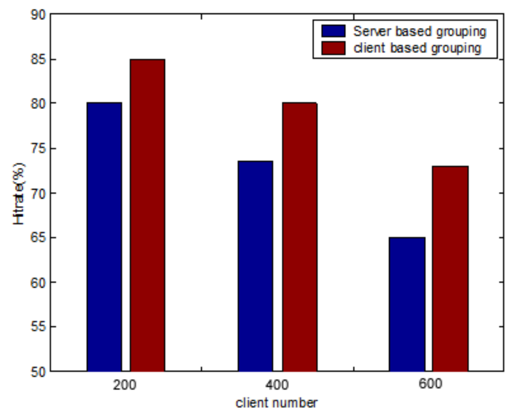


Figure 8 Server-oriented grouping vs. client-oriented grouping, cache size=400

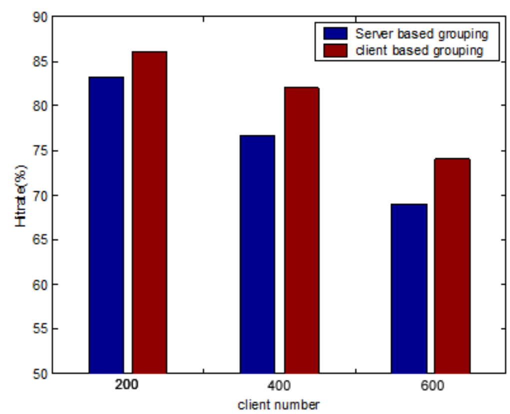


Figure 9 Server-oriented grouping vs. client-oriented grouping, cache size=750

B. Simulation framework

In order to obtain the pure prefetching effect, we first experiment on local machine that only consists of local cache and local disk. The prefetching result in local client can directly influence the performance of the whole system. Figure 10 shows the hit rate of several prefetching algorithms.

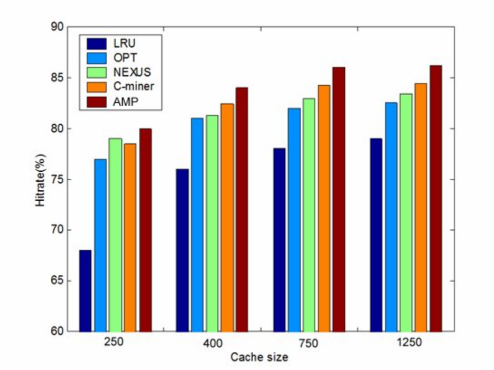
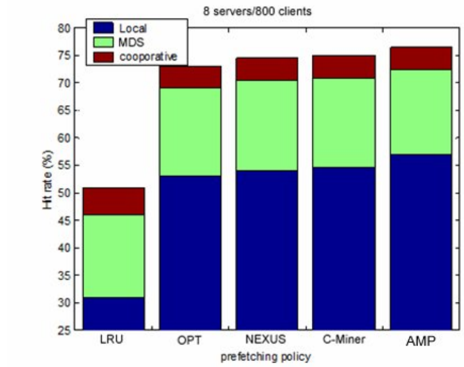


Figure 10 Client local hit rate (HP trace).

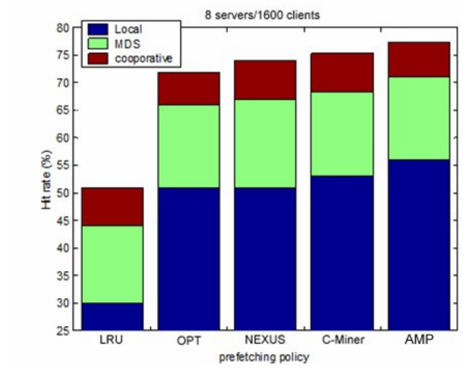
In order to simulate a distributed storage system, we develop a system simulator to study the clustered-MDS based storage system. In our simulation framework, the storage system consists of four layers: 1) client cache, 2) metadata server cache, 3) cooperative cache, and 4) metadata server hard disks. When one client needs to obtain a file metadata, it first checks its local cache (client cache). Upon a cache miss, the client sends the request to the corresponding MDS, a corresponding network latency would be added. Since our main goal is to explore the distributed storage system and prefetching, we assume that all nodes are connected with a network delay of 0.3 ms; if the MDS also sees a miss, the MDS looks up the cooperative cache, which would add another network latency. Otherwise, MDS can only fetch the metadata files from the disk, which potentially experiences a relatively higher delay due to large disk access.

C. Hit rate comparison

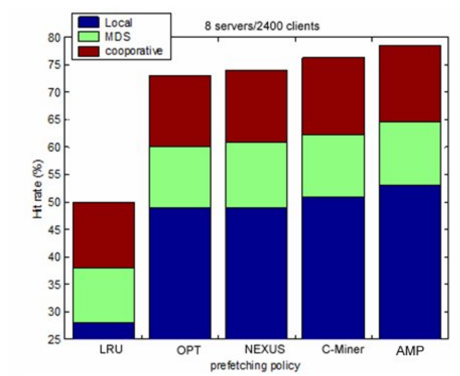
The overall cache hit rate includes three components: client local hit, metadata server memory hit, and cooperative cache hit. Obviously, local hit rate directly reflects the effectiveness of the prefetching algorithm because the prefetching algorithm is executed in this layer. We have collected the hit rate for all these three levels. The client cache is the most important part, because it directly reflects the effect of prefetching and greatly influences the hit rate and response time. Figure 11 shows the hit rate when the system contains different clients. It shows that AMP always has the best local hit rate, which is uniform with the local hit rate experiment. Also, we can see that three prefetching algorithm AMP, C-miner and Nexus all beat the off-line optimal cache replacement algorithm (OPT) that doesn't perform prefetching. This proves the effectiveness of metadata prefetching. .



System hit rate with 8MDS and 800 clients (HP trace).

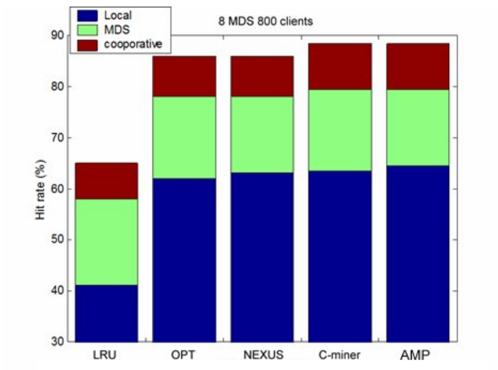


System hit rate with 8MDS and 1600 clients(HP trace).

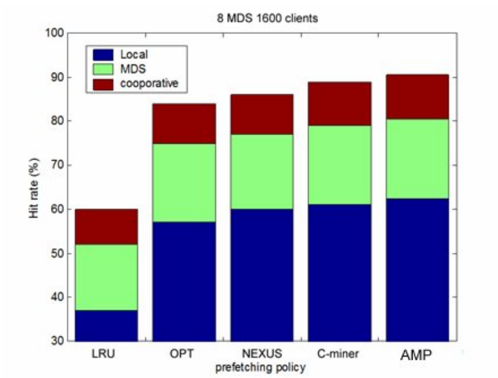


System hit rate with 8MDS and 2400 clients (HP trace).

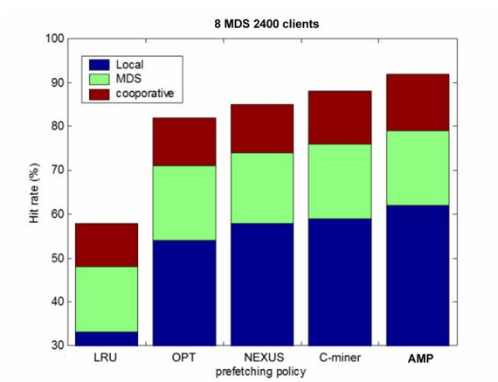
Figure 12



System hit rate with 8MDS and 800 clients (Harvard SOS trace).



System hit rate with 8MDS and 1600 clients (Harvard SOS trace).



System hit rate with 8MDS and 2400 clients (Harvard SOS trace).

Figure 13

D. Response time Comparison

The average response time is measured by using Disksim. As explained earlier, the whole system has four layers, including client cache, MDS cache, cooperative cache and MDS disk. From Figure 14 and Figure 15, we can see that AMP has the best response time. Compared with LRU, NEXUS and C-miner, trace-driven simulations show that AMP can improve the hit rates by up to 12%, 4.5% and 4%, respectively, while reduce the average response time by up to 60%, 12% and 8%, respectively.

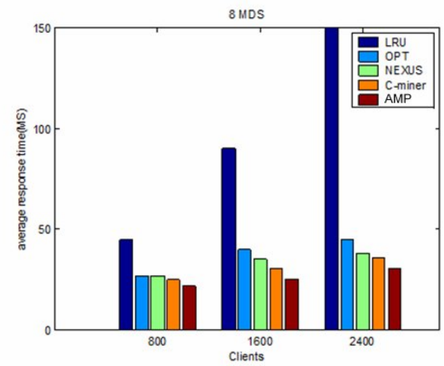


Figure 14 Average response time for 8 MDS (HP traces)

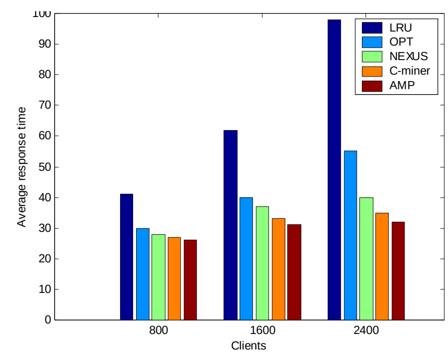


Figure 15 Average response time for 8 MDS (Harvard SOS traces)

5. Conclusion

This paper proposes an Affinity-based Metadata Prefetching (AMP) Scheme for distributed large-scale storage systems. By exploiting the past affinities between file metadata, AMP can achieve aggressive but efficient prefetching. AMP has following contributions:

- By precisely analyzing the past requests, AMP can discover deeper and more accurate metadata affinities.
- AMP takes advantages some the small-size characteristic and performs more aggressive prefetching than state-of-the-art prefetching algorithms.
- AMP has small overhead and can be implemented as an online prefetching algorithm.

Both theoretical analytical and simulation results improve the cache hit rate and reduce metadata time significantly.

References

[1] "Lustre: A scalable, high-performance file system," Cluster File Systems Inc. white paper, version 1.0, Nov. 2002.

[2] Y. Zhu, H. Jiang, and J. Wang, "Hierarchical bloom filter arrays (hba):a novel, scalable metadata management system for large cluster-based storage," in Cluster Computing, 2004 IEEE International Conference on, 2004, pp. 165–174.

[3] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur, "PVFS: A parallel file system for linux clusters," in Proceedings of the 4th Annual Linux Showcase and

Conference. Atlanta, GA: USENIX Association, 2000, pp. 317–327.
[4] S. Ghemawat, H. Gobioff, and S.-T. Leung, “The google file system,” in SOSP, 2003, pp. 29–43.
[5] L.-F. Cabrera and D. D. E. Long. Swift: Using distributed disk striping to provide high I/O data rates. Computing Systems, 4(4):405–436, 1991.
[6] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. In Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 92–103,San Jose, CA, Oct. 1998.
[7] J. H. Hartman and J. K. Ousterhout, “The Zebra striped network file system,” in High Performance Mass Storage and Parallel I/O:Technologies and Applications, H. Jin, T. Cortes, and R. Buyya, Eds. New York, NY: IEEE Computer Society Press and Wiley, 2001, pp.309–329.
[8] E. J. Otoo, D. Rotem, and A. Romosan, “Optimal file-bundle caching algorithms for data-grids,” in SC’2004 Conference CD. Pittsburgh, PA: IEEE/ACM SIGARCH, Nov. 2004, lBNL.
[9] M. Gupta and M. Ammar, “A novel multicast scheduling scheme for multimedia servers with variable access patterns,” Dec. 26 2002.
[10] A. Muthitacharoen, B. Chen, and D. Mazieres, “A low-bandwidth network file system,” in SOSP, 2001, pp. 174–187.
[11] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller, “Dynamic metadata management for petabyte-scale file systems,” in SC’2004Conference CD. Pittsburgh, PA: IEEE/ACM SIGARCH, Nov. 2004,
[12] D. Roselli, J. Lorch, and T. Anderson. A comparison of file system workloads. In Proceedings of the 2000 USENIX Annual Technical Conference, pages 41–54, June 2000.
[13] D. Roselli, J. Lorch, and T. Anderson. A comparison of file system workloads. In Proceedings of the 2000 USENIX Annual Technical Conference, pages 41–54, June 2000.
[14] P. Gu, Y. Zhu, H. Jiang, and J. Wang, “Nexus: A novel weightedgraph-based prefetching algorithm for metadata servers in petabyte-scale storage systems,” Proc. 6th IEEE Int’l Symp. on Cluster Computing and the Grid, pp. 409–416, 2006.
[15] P. Cao, E. Felten, and K. Li. Application-controlled file caching policies. In USENIX Summer 1994 Technical Conference, pages 171–182, June 1994.
[16] P. Cao, E.W. Felten, A. Karlin, and K. Li. A study of integrated prefetching and caching strategies. In Proceedings of ACM SIGMETRICS, May 1995.
[17] A. Tomkins, R. H. Patterson, and G. Gibson. Informed multi-process prefetching and caching. In Proceedings of the 1997 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pages 100–114. ACM Press, 1997.
[18] T. Kimbrel, A. Tomkins, R. H. Patterson, B. Bershad, P. Cao, E. Felten, G. Gibson, A. R. Karlin, and K. Li. A trace-driven comparison of algorithms for parallel prefetching and caching. In Proceedings of the 1996 Symposium on Operating Systems Design and Implementation, pages 19–34. USENIX Association, 1996.
[19] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In the 15th ACM Symposium on Operating System Principles, 1995.
[20] V. Soloviev. Prefetching in segmented disk cache for multi-disk systems. In Proceedings of the fourth workshop on I/O in parallel and distributed systems, pages 69–82. ACM Press, 1996.
[21] A. D. Brown, T. C. Mowry, and O. Krieger. Compiler-based I/O prefetching for out-of-core applications. ACM Transactions on Computer Systems, 19(2):111–170, 2001.
[22] T. C. Mowry, A. K. Demke, and O. Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. In Proceedings of the 1996 Symposium on Operating Systems Design and Implementation, pages 3–17. USENIX Association, Oct. 1996.
[23] F. W. Chang and G. A. Gibson. Automatic I/O hint generation through speculative execution. In Operating Systems Design and Implementation, pages 1–14, 1999.
[24] G. Kuenning. Design of the SEER predictive caching scheme. In Workshop on Mobile Computing Systems and Applications, 1994.
[25] G. H. Kuenning and G. J. Popek. Automated hoarding for mobile computers. In Proceedings of the 15th Symposium on Operating Systems Principles, pages 264–275, St. Malo, France, Oct. 1997. ACM.
[26] T. M. Kroeger and D. D. E. Long. Predicting file-system actions from prior events. In 1996 USENIX Annual Technical Conference, pages 319– 328, 1996.
[27] H. Lei and D. Duchamp. An analytical approach to file prefetching. In 1997 USENIX Annual Technical Conference, Anaheim, California, USA, 1997.
[28] C. D. Tait, H. Lei, S. Acharya, and H. Chang. Intelligent file hoarding for mobile computers. In Mobile Computing and Networking, pages 119–125, 1995.
[29] J. Han and M. Kamber. Data Mining: Concepts and Techniques. Morgan Kaufmann Publishers, 2001.
[30] Z. Li, Z. Chen, S. M. Srinivasan, and Y. Zhou. C-miner: Mining block correlations in storage systems. In Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST), pages 173–186, 2004.
[31] Peng Gu, Yifeng Zhu, Hong Jiang, Jun Wang, Nexus: A Novel Weighted-Graph-Based Prefetching Algorithm for Metadata Servers in Petabyte-Scale Storage Systems. International Symposium on Cluster Computing and the Grid, 2006.
[32] Shuang Liang, Song Jiang, Xiaodong Zhang. STEP: Sequentiality and Thrashing Detection Based Prefetching to Improve Performance of Networked Storage Servers. In Proceedings of the ICDCS'07, Toronto, Canada, June 2007.
[33] G. Ganger. Systemoriented evaluation of I/O subsystem performance. Technical Report CSE-TR-243-95, University of Michigan, June 1995.
[34] E. Riedel, M. Kallahalla, and R. Swaminathan, “A framework for evaluating storage system security,” in FAST, 2002, pp. 15–30.
[35] “SOS Project Traces,” [online]. Available: http://www.eecs.harvard.edu/sos/traces.html.
[36] Daniel Ellard and Margo Seltzer, New NFS Tracing Tools and Techniques for System Analysis, 2003 LISA.