Nexus: A Novel Weighted-Graph-Based Prefetching Algorithm for Metadata Servers in Petabyte-Scale Storage Systems

Peng Gu, Yifeng Zhu*, Hong Jiang, Jun Wang
Computer Science and Engineering
University of Nebraska-Lincoln Lincoln, Nebraska 68588
Email:{penggu,jiang,wang}@cse.unl.edu
*Electrical and Computer Engineering
University of Maine Orono, ME 04469-5708
Email: zhu@eece.maine.edu

***Abstract*— An efficient, accurate and distributed metadata-oriented prefetching scheme is critical to the overall performance in large distributed storage systems. In this paper, we present a novel weighted-graph-based prefetching technique, built on successor relationship, to gain performance benefit from prefetching specifically for clustered metadata servers, an arrangement envisioned necessary for petabyte-scale distributed storage systems. Extensive trace-driven simulations show that by adopting our new prefetching algorithm, the hit rate for metadata access on the client site can be increased by up to 13%, while the average response time of metadata operations can be reduced by up to 67%, compared with LRU and an existing state of the art prefetching algorithm.**

I. Introduction

A novel decoupled storage architecture diverting actual file data flows away from metadata traffics has emerged to be an effective approach to alleviate the I/O bottleneck in modern storage systems [1]–[4]. Unlike conventional storage systems, these new storage architectures use separate servers for data and metadata services, as shown in Fig 1. Accordingly, large

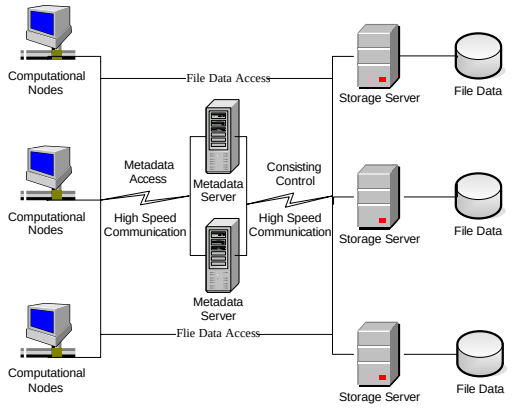


Fig. 1. System architecture

volume of actual file data does not need to be transferred through metadata servers, which significantly increases the data throughput. Previous studies on this new storage architecture mainly focus on optimizing the scalability and efficiency of file data accesses by using a RAID style striping [5], caching [6], scheduling [7] and networking [8] and very little attention has been paid to the scalability of the metadata management. However, the performance of metadata services plays a critical role in achieving high I/O scalability and throughput, especially in light of the rapidly increasing scale in modern storage systems where the volume of data reaches and even exceeds Peta bytes (10^{15} bytes) while metadata amounts to Tera bytes (10^{12} bytes) or more [9]. In fact, more than 50% of all I/O operations are to metadata [10], suggesting further that multiple metadata servers are required for a petabyte-scale storage system to avoid potential performance bottleneck on a centralized metadata server. This paper takes advantages of some unique characteristics of metadata and proposes a new prefetching scheme for metadata access that is able to scale up the performance of metadata services in large scale storage systems.

By exploiting the access locality widely exhibited in most I/O workloads, caching and prefetching have become an effective approach to boost I/O performance by absorbing a large number of I/O operations before they touch disk surfaces. However, prefetching metadata for large metadata services imposes two challenges. First, the prefetching operation has to be conducted in a distributed environment. Due to the skewed load toward metadata in most file systems [11], a centralized metadata management system will not scale up well with the I/O workload in a large scale storage system [2], [9]. Accordingly, in a petabyte-scale storage system, the load of metadata services is likely distributed among a group of metadata servers. Second, existing caching and prefetching algorithms may not work well for metadata. Most caching and prefetching schemes are designed for and apply on actual file data and ignore metadata characteristics. Those algorithms are not specifically optimized for metadata accesses since usually file data and metadata operations show different characteristics and exhibit different access behaviors. For example, a file might be read multiple times while its metadata is only accessed once. A “ls -l” command touches the metadata of multiple files but might not access their data. In addition, the size of metadata is typically uniform and much smaller than the size of file data in most file systems. In order to achieve optimal performance, a new prefetching and caching algorithm that considers the differences between data and metadata is clearly desirable.

The most important characteristic of metadata is its relative small size compared with typical file sizes. With a relatively small data size, the mis-prefetching penalty for metadata 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. In contrast, most of the previous prefetching algorithms share the same characteristic in that they are conservative on prefetching. That is, they prefetch at most one file upon each cache miss. In addition, even when a cache miss happens, certain rigid policies are applied before issuing a prefetching in order to maintain a high level of prefetching accuracy. Nevertheless, considering the huge number and the relatively small size of metadata items, aggressive prefetching can be profitable provided that a higher system performance and a reasonable prediction accuracy is achieved.

In this paper, we develop a novel prefetching algorithm named Nexus to perform more aggressive prefetching while maintaining a reasonable prefetching accuracy. A looking ahead history window is deployed to capture better locality and to scrutinize the real successor relationship among interleaved accesses sequence. Comprehensive simulation results indicate that Nexus significantly improves the performance of metadata retrieval in Peta-byte scale cluster storage system.

The outline of the rest of the paper is as follows: Related work is discussed in Section II. Section III described our Nexus algorithm in detail. Evaluation methodologies and results are discussed in section IV. We conclude this paper in section V.

II. Related Work

Previous research work on prefetching both at the disk level and at the file level can be classified into three categories: predictive prefetching [12], application-controlled prefetching [13], and compiler-directed prefetching [14].

Among the latest advancement in the area of predictive prefetching, in order to study the prefetching accuracy while maintaining a reasonable performance gain, Darrell Long et al. introduced several file access predictors including First Successor, Last Successor, Noah (Stable Successor) [15], Recent Popularity (also known as Best j-out-of-k) and Probability-based Successor Group Prediction [16], [17]. The differences among these predictors are summarized as follows.

a) First Successor: The file that followed file *A* the first time *A* was accessed is always predicted to follow *A*.

b) Last Successor: The file that followed file *A* the last time *A* was accessed is predicted to follow *A*.

c) Noah (Stable Successor): Similar to Last Successor, except that a current prediction is maintained; and the current prediction is changed to last successor if last successor was the same for *S* consecutive accesses where *S* is a predefined parameter.

d) Recent Popularity (Best j-out-of-k): Based on last k observations on file *A*'s successors, if j out of those k observations turn out to target the same file *B*, then *B* will be predicted to follow *A*.

e) Probability-based Successor Group Prediction: Based on file successor observations, a file relationship graph is built to represent the probability of a given file following another. Based on the relationship graph, the prefetch strategy builds the prefetching group by following steps:

1) The missed item is first added into the group.
2) Add the items with the highest conditional probability under the condition the items in the current prefetching group were accessed together.
3) Repeat step 2 until the group size limitation is met.

III. Nexus: A Weighted-Graph-Based Prefetching Algorithm

As a more effective way for metadata prefetching, our Nexus algorithm distinguishes itself in two aspects. First, Nexus can more accurately capture the metadata access temporal locality exhibited in metadata access streams by observing the affinity among both immediate and indirect successors. Second, Nexus exploits the fact that metadata usually is small in size and deploy an aggressive prefetching strategy.

A. Relationship graph overview

Our algorithm uses a metadata relationship graph to assist prefetching decision making. The relationship graph is used to dynamically represent the locality strength between predecessors and successors in metadata access streams. Directed graphs are chosen to represent the relationship since the relationship between a predecessor and a successor is essentially unidirectional. Each metadata corresponding to a file or directory is represented as a vertex in our relationship graph. The locality strength between a pair of metadata items is represented as a weighed edge. Figure 2 shows an example of relationship graph consisting of metadata for six files/directories. From this graph, we can observe that the predecessor-successor relationship between $/usr$ and $/usr/bin$ is much stronger than that between $/usr$ and $/usr/src$.

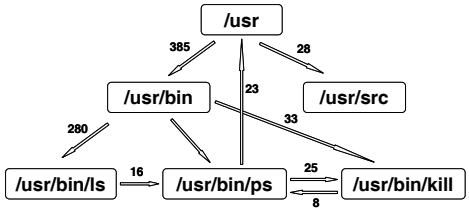


Fig. 2. Relationship graph demo

B. Relationship graph construction

To understand how this relationship graph works for improved prefetching performance, it is necessary to first understand how this graph is built. The relationship graph is built on the fly while the MDS receives and serves requests from a large number of clients. A history window with a predefined capacity is used to keep the requests most recently received by the MDS server. For example, if the history window capacity is set to ten, only ten most recent requests are kept in the window. Upon the arrival of a new request, the oldest request in this history window is replaced by the new comer. In this

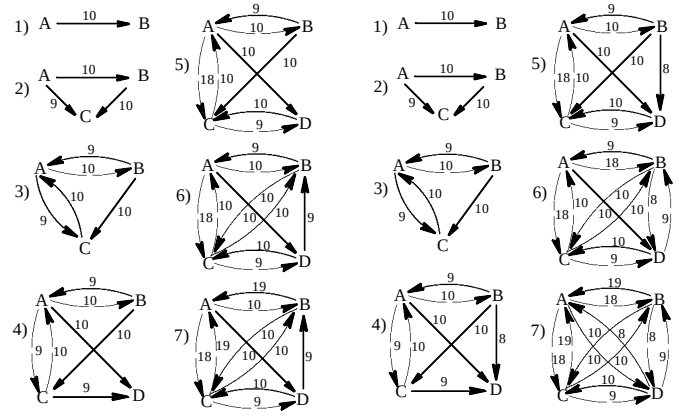
way the history window is dynamically updated and always contains the current predecessor-successor relationship at any time. The relationship information is then integrated into the graph on a per-request basis, by either inserting a new edge (if the predecessor-successor relationship is discovered for the very first time) into the graph or adding an appropriate weight to an existing edge (if this relationship has been observed before). A pseudocode describing how a relationship graph is built is presented below.

```
// Let G denote the graph to be built
BUILD-RELATIONSHIP-GRAPH(G)
1  G← ∅
2  for each new incoming metadata request j
3     foreach metadata request i (i ≠ j) in history window
4        if edge (i, j) ∉ G
5        then add an edge (i, j) to G with appropriate weight
6        else add appropriate weight to edge (i, j)
7     replace the oldest item in history window with j
```

For example, if the history window size is two and a request sequence of

$$ABCADCBA\cdots$$

is observed, the step-by-step graph construction from scratch is shown in Figure 3(a) (The weight assignment methodology taken here is linear decremental, described later in Section III-E.1 on page 4). In contrast, Figure 3(b) shows the same graph construction procedure with a history window size of three.



(a) Looking ahead window size = 2 (b) Looking ahead window size = 3

Fig. 3. Graph construction examples

C. Prefetching based on the relationship graph

Once the graph is built for the access sequence $ABCADCBA\cdots$ as shown in Figure 3(a) and Figure 3(b), it is possible to prefetch a successor group with an adjustable size in the graph when a cache miss happens for an element in that group. The prediction result depends on the order of the outbound edge weights (represented by values associated with arrows in the relationship graph) of the latest missed element. A larger weight indicates a stronger relationship and a higher prefetching priority. In the above sample access sequence, supposing that the last request A sees a miss, according to the graph shown in Figure 3(a), the prediction result will be $\{C\}$ if the prefetching group size is one, or $\{C, D\}$ if the prefetching group size is two; similar results deduced from Figure 3(b) will be $\{B\}$ and $\{B, C\}$, respectively.

D. Major advantages of Nexus

1) The farther the sight, the wiser the decision: The key difference between the relationship-based and probability-based approaches lies in the ability to look farther than the immediate successor. The shortcoming of the probability-based prefetching model is obvious: it only considers the immediate successors as candidates for future prediction. As a consequence, any successors after the immediate successor are ignored. This short-sighted method is incapable of identifying the affinity of two references with some intervals, which widely exists in many applications. For example, for the pattern "$A?B$", we can easily find two situations where this pattern exhibits.

- Compiling programs: gcc compiler("A") is always first launched; and then the source code("$?$") to be compiled is loaded; at last the common header files or common shared libraries ("B") is loaded afterward.
- Multimedia application: initially media player application ("A") is launched; after that the media clip ("$?$") to be played is loaded; at last the decoder program ("B") for that type of media is loaded.

In addition to above mentioned applications, interleaved application may create similar kinds of scenarios. The probability-based model cannot detect such access patterns, thus limiting its ability to make better predictions. However, this omitted information is considered in our relationship-based prefetching algorithm, which is able to look farther than the immediate successor when we build our relationship graph.

We use a simple trace sequence mentioned before, $ABCADCBA\cdots$, to further illustrate the difference between the probability-based approach and our relationship-based method. In the probability-based model, since C never appears immediately after A, C will never be predicted as A's successor. In fact, the reference stream shows that C is a good candidate as A's successor because it always shows up *next next* to A. The rationale is that the pattern we observed is a repetition of pattern "$A?C$" and we assume this pattern will repeat in the near future. As discussed in III-C, should our relationship-based prediction be applied, three out of four prediction results will contain C.

From the above example, we clearly see the advantages of relationship-based prefetching over probability-based prefetching. The essential ability to look farther than the immediate successor directly renders this advantage. In contrast, should we apply the same idea to probability-based approach, the complexity of the algorithm would increase exponentially. For example, if looking ahead window size is set to increased from 1 to 2, using probability-based approach, we would have to maintain the conditional probability for each triple $P(C|AB)$ instead of for each two-tuples $P(B|A)$.

2) Aggressive prefetching is natural for metadata servers: All previous prefetching algorithms tend to be conservative due to the prohibitive mis-prefetch penalty and cache pollution. However, the penalty of an incorrect metadata prefetch might be much less prohibitive than that of the file data

prefetch, and the cache pollution problem is not as severe as in the case of file data caching. The evidence behind this reasoning is the observation that while the average file size is 22KB [18], the average size of a file's metadata is only 1.375KB [1]. This observation encourages us to conduct more aggressive prefetching on metadata.

E. Algorithm design considerations

In implementing our algorithms, several design factors need to be considered to optimize the performance. Corresponding sensitivity studies on those factors are carried out as follows.

1) Successor relationship strength: Assigning an appropriate weight between the nodes to represent the strength of their relationship as predecessor and successor is critical to our algorithm because it affects the prediction accuracy of our algorithm. A formulated description of this problem is: Given an access sequence of length n:

$$M_1 M_2 M_3 \ldots M_n,$$

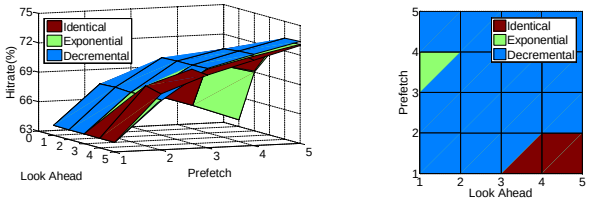
how much weight should be added to the predecessor-successors edges,

$$(M_1, M_2), (M_1, M_3), \ldots, (M_1, M_n),$$

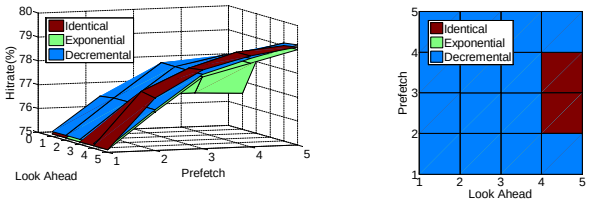
respectively. Four approaches are taken into consideration:

- *Identical assignment*
  Assigning all the successors of M_1 the same importance. This approach is very similar to the probability model introduced by Griffioen and Appleton [19]. It may look simple and straightforward, but it is indeed effective. The key point is that at least the successor following the immediate successors are taken into consideration. However, the draw back of this approach is also obvious: it cannot differentiate the importance of the immediate successor and its followers, which might subsequently skew the relationship strengths to some extend. This approach is referred to as *identical* assignment for later discussions.
- *Linear decremental assignment* The assumption behind this approach is that the closer the access distance in the reference stream, the stronger the relationship. For example, we may assign those edge weights mentioned above in a linear decremental order, as 10 for (M_1, M_2), 9 for (M_1, M_3), 8 for (M_1, M_4), and so on. (The weight in the example shown in Figure 3(a) and 3(b) is calculated this way.) This approach is referred to as *decremental* assignment in the rest of this paper.
- *Polynomial decremental assignment*
  Another possibility is that, with increase in the successor distance, the decrease in the relationship strength might be more radical than the linear one. For example, polynomial decrement assignment is a possible alternative solution. This assumption is based on the observation of the attenuation of radiation in the air in our real life.
- *Exponential decremental assignment*
  The attenuation of edge weights might be even faster than polynomial decrement. In this case, an exponential decrement model is adopted. This approach is referred to as *exponential* decremental assignment in the future.

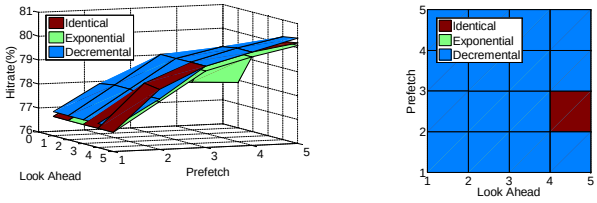
To find out which assignment method can best reflect the locality strength in the metadata reference streams, we conduct experiments on the HP trace [10] to compare the hit rate achieved by those four edge-weight assignment methods. To be comprehensive, these experiments are conducted with different configurations in three dimensions: cache size, number of successors to look ahead (or history window size), and number of successors to prefetch as a group (or prefetching group size). Since the result for the polynomial assignment is very close to that for the exponential assignment, we remove the former results to show readers a clearer figure. The results for the remaining three approaches are shown in Figure 4.



(a) Cache size = 250



(b) Cache size = 750

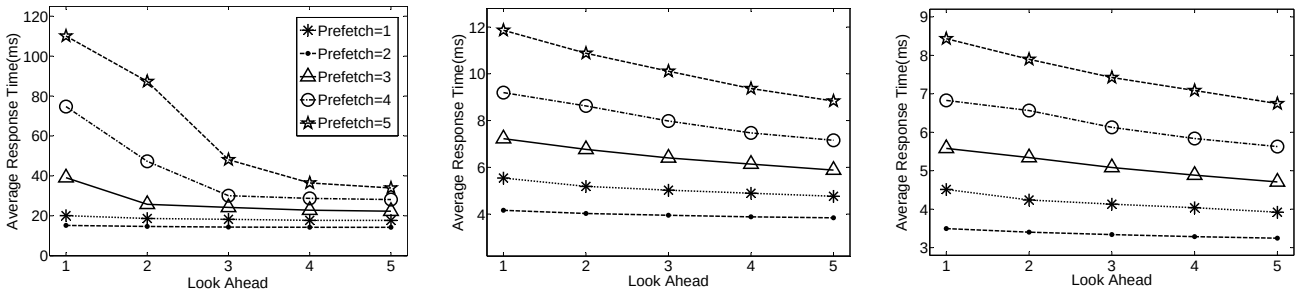


(c) Cache size = 1250

Fig. 4. Edge weight assignment approaches comparison

In Figure 4, the 3D graphs on the left show the hit rate achieved by those three approaches over three cache size (in terms of cachelines) configurations (i.e. 250, 750 and 1250) with both the look-ahead window size and prefetching group-size varying from 1 to 5. (These values are carefully chosen in order to be representative while non-exhaustive.) The three 2D graphs on the right show the corresponding planform (a X-Y plane looking downward along the Z axis) of the same measurements. These 2D graphs clearly show that the linear *decremental* assignment approach takes the lead most of the time. We also notice that the identical assignment beats others

[1]The calculation is carried out as following: Since the average file size is 22KB, and an inode of 128 bytes is allocated for every 2KB of file data [18]. Thus the average size of inode for each file is $22KB/2KB \times 128Bytes = 1.375KB$. It means that the size of metadata is roughly 6.25% of the corresponding file data.



(a) Cache Size = 250 (b) Cache Size = 750 (c) Cache Size = 1250

Fig. 5. Sensitivity study: look ahead windows size and prefetch group size

in some cases even though this approach is very simple. The linear decremental assignment approach consistently outperforms others. Thus, in the future experiments, we will deploy this approach as our edge-weight-assignment scheme.

2) How far to look ahead and how many to prefetch: To fully exploit the benefit of bulk prefetching, we need to decide the distance to look ahead and the bulk size to prefetch. Looking ahead too far may compromise the algorithm's effectiveness by introducing noises to the relationship graph; and prefetching too much may result in a lot of inaccurate prefetching, possible cache pollution, and cause performance degradation. We compare the average response time by performing a number of experiments on a combination of these two key parameters, i.e., look ahead window size and prefetching group size. The result is shown in Figure 5. From Figure 5, we found that looking ahead 5 successive files' metadata and prefetching 2 files' metadata at a time turned out to be the best combination. The results also seem to suggest that the larger the looking ahead window size, the better the hit rate achieved. This observation prompts us to experiment on much larger look-ahead window sizes, with sizes 10, 50, and 100 respectively, and found contradicting results to our conjecture: none of those three look-ahead window size configurations achieves a better hit rate than the windows size of 5. The reason is that looking too far ahead might overwhelm the prefetching algorithm by introducing too much noise–those irrelevant future accesses are also taken into consideration as successors, reducing the usefulness of the relationships captured by the look-ahead window. Hit rate comparison reveals consistent results but are omitted here due to space limitation. In the rest of the paper's experiments, the look-ahead distance and the prefetching group size are fixed to 5 and 2 respectively for best performance gains.

3) Server-oriented grouping vs. client-oriented grouping: One way to improve the effectiveness of the metadata relationship graph is to enforce better locality. Since multiple clients may access any given metadata server simultaneously, most likely request streams from different clients will be interleaved, making the pattern more difficult to observe. Thus it may be a good idea to differentiate the different clients when building the relationship graph. Thus there are two different approaches to build the relationship graph: 1) Build a relationship graph for all the requests received by a particular metadata server; or 2) Build a relationship graph for requests sent from each individual client and received by a particular metadata server. In this paper, we refer to the former version as server-oriented access grouping, and the latter as client-oriented access grouping.

We have developed a client-oriented grouping algorithm and compared it with the server-oriented grouping by running them on the HP traces, as shown in Figure 6. r Figure 6

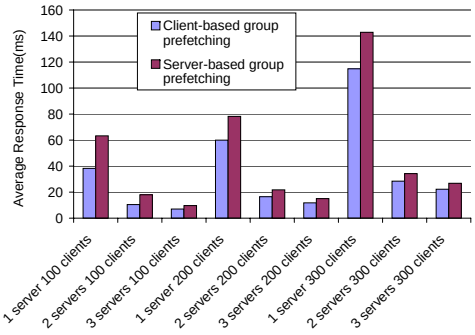


Fig. 6. Server-oriented grouping *vs client-oriented grouping*

clearly shows that client-oriented grouping algorithm always outperforms the server-oriented one. Thus we adopt the client-oriented grouping algorithm whenever possible.

IV. Evaluation Methodology and Results

A. Workloads

We evaluate our design by running trace-driven simulations over the LLNL trace collected at Lawrence Livermore National Laboratory in July 2003 [20] and the HP file system trace collected at the HP Lab in December 2001 [21]. These traces gather I/O events of both file data and metadata. In our simulations, we filter out the file data activities and feed only metadata events to our simulator.

1) LLNL trace: One of the main reasons for petabyte-scale storage systems is the need to accommodate scientific applications that are increasingly demanding on I/O and storage capacities and capabilities. As a result, some of the best traces to evaluate our prefetching algorithm are those generated by scientific applications. To the best of our knowledge, the only recent scientific application trace publicly available for large

clusters is the LLNL2003 file system trace. It was obtained in the Lustre Lite [1] parallel file system on a large Linux cluster with more than 800 dual-processor nodes. It consists of 6403 trace files with a total of $46,537,033$ I/O events. In our simulations, we only consider the metadata activities, described in Figure 7. The metadata operations are further

Name	Count	Description
access	16	check user's access permissions
close	111,215	close a file descriptor
fstat64	81,663	retrieve file status
ftruncate64	198	truncate a file to a specified length
open	327,990	open or create a file
stat64	59,892	display file status
statfs	980	display file system status
unlink	8	delete a name and possibly the file it refers to

Fig. 7. List of operations obtained by *strace* in LLNL trace collection

classified into two categories: metadata read and metadata write. Operations such as $access$, and $stat$ fall into the metadata read group, while $ftruncate64$ and $unlink$ belong to the metadata write group since they need to modify the attributes of the file. However, the classification of $open$ and $close$ remains ambiguous. An $open$ operation cannot be simply classified as metadata read since it may create files according to its semantics in UNIX. Similarly, a $close$ operation can be classified into both groups since it may incur metadata update operations, depending on whether the file attributes are dirty or not. For $open$ requests, the situation is easier since we can look at the parameter and return value of the system call to determine its type. For example, if the parameter is O_RDONLY and the return value is a positive number, then we know for sure that this is a metadata read operation. For $close$, we can always treat it as a metadata write assuming that the *last modify time* field is always updated upon file closure.

2) HP trace: To provide a more comprehensive comparison, we also conduct our simulations on the HP trace [21], a 10-day trace of file system collected on a time-sharing server with a total of 500 GB storage capacity and 236 users. Since it is not a scientific application trace, we artificially scale it up to emulate a multi-MDS multi-client application by merging multiple trace files into one to increase the access density while maintaining timing order of the access sequences. In our simulations, any I/O operations not related to metadata are also filtered out.

B. Simulation framework

A simulation framework was developed to simulate a clustered-MDS based storage system consisting of multiple MDSs and multiple clients with the ability to adopt flexible caching/prefetching algorithms. In such a hierarchical storage system, metadata consistency control becomes a prominent problem for the designers. However, this is not the focus of our current study, which is the design and evaluation of a novel prefetching algorithm for metadata. To simplify our simulation design, cooperative caching [22], a widely used hierarchical cache design, together with its cache coherence control mechanism, i.e. write-invalidate [23], is adopted in our simulation framework to cope with the consistency issue. However, it must be noted that the choice of cooperative caching is pragmatic for its relative maturity and simplicity and, as such, it does not necessarily imply that it is the only or best choice for consistency control. In fact, we believe that a metadata-oriented consistency protocol is needed to optimize the performance, which is one of our future research directions.

In our simulation framework, the storage system consists of four layers: 1) client cache, 2) metadata server cache, 3) cooperative cache, and 4) hard disks. When the system receives a metadata request, it first checks its local cache (client cache); upon an cache miss, the client sends the request to the corresponding MDS; if the MDS also sees a miss, the MDS looks up the cooperative cache as a last resort before sending the request to disks.

Thus the overall cache hit rate includes three components from the client's point of view: local hit (client cache hit), remote hit (metadata server cache hit), and cooperative cache hit. Obviously, local hit rate directly reflects the effectiveness of the prefetching algorithm because grouping and prefetching are done on the client site.

If, in the best case, a metadata request is satisfied by the client cache, the response time for that request is estimated as local main memory access latency. Otherwise, if that request is sent to a MDS and satisfied by the server cache, the overhead of network delay is included in the response time. In an even worse case, the server cache does not contain the requested metadata while the cooperative cache does, extra network delay should be considered. In the worst case, the MDS has to send the request to disks where the requested metadata resides, extra disk access overhead contributes to the response time.

Prefetching happens when a client sees a local cache miss. In this case the metadata request is sent to MDS. Upon arrival of that request at the metadata server, the requested metadata is retrieved (from server side cache, cooperative cache or hard disk) by the MDS along with the entire prefetching group.

C. Trace-driven simulations

Trace-driven simulations based on the HP trace and the LLNL trace were conducted to compare different caching-prefetching algorithms, including conventional caching algorithms such as LRU (Least Recently Used), LFU (Least Frequently Used) and MRU (Most Recently Used), some prefetching algorithms such as First Successor, Last Successor, and state of the art prefetching algorithms such as Noah (Stable Successor), Recent Popularity (also known as Best j-out-of-k), and Probability-Graph Based prefetching (referred to as DL in the rest of this paper).

Most previous studies use only prediction accuracy to evaluate the prefetching effectiveness. However, this measurement is neither adequate nor sufficient. The ultimate goal

of prefetching is to reduce the average response time by absorbing I/O requests before they reach disks. A higher prediction accuracy does not necessarily indicate a higher hit rate nor a lower average response time, since too conservative prefetching, even with a high level of prefetching accuracy, might not be as beneficial. Thus in our simulations, we not only measure the cache hit rate, but also the average response time by integrating a golden disk simulator, DiskSim 3.0 [24], into our simulation framework.

We conduct experiments for all the caching/prefetching algorithms mentioned above. Due to space limitation, we remove the results for less representative algorithms, including LFU, MRU (these two are always worse than LRU); First Successor, Last Successor, Noah, Recent Popularity, since these algorithms are consistently inferior to DL. In addition, Optimal Caching [25], referred to as OPT in the rest of this paper, is simulated as an ideal caching algorithm for theoretical comparison purpose. In OPT, the item to be replaced is always the farthest in the future access sequence. Since the prefetching group size for Nexus is set to 2, we have tried both 1 and 2 for this parameter on DL, referred to as DL1 and DL2, respectively, in order to provide a fair comparison. In sum, in this paper we will present the results for five caching/prefetching algorithms including Nexus, DL1, DL2, LRU and OPT.

In addition, we also conducted these experiments in a multi-metadata-server multi-client environment in order to test the scalability of our algorithm.

D. Hit rate comparison

We have collected the hit rate for all the three levels of cache: client cache, server cache and cooperative cache. Since a group of metadata are prefetched to a client cache upon a cache miss, and the server cache and the cooperative cache might offset the effect of the client cache hit rate, the overall hit rate does not show fully and truly the merits of our prefetching algorithm. Instead, it is the client cache that enjoys the benefits from our prefetching algorithm. The experimental results confirmed our suspicion. Figure 8 shows that the client cache hit rate comparisons on the HP trace and the LLNL trace. Due to space limitation, only the results

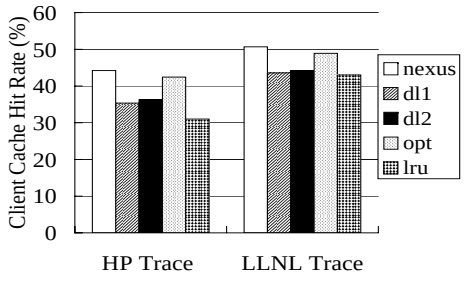


Fig. 8. Hit rate comparison with 2 metadata servers and 200 clients

for 2 metadata servers and 200 clients are presented in this paper.Figure 8 shows that the client cache hit rate of Nexus can outperform DL1 and DL2 by more than 8%. Furthermore, Nexus even beats OPT [2] by a small margin. These results suggest that Nexus can effectively identify the locality strength and make a judicious prefetching decision without noticeable cache pollution side-effect.

E. Average response time comparison

To measure the prefetch overhead of Nexus, the average response time is measured by incorporating disk simulators. Figure 9 presents the results obtained from both HP trace and LLNL trace. Again, Nexus achieves consistently better performance than any other algorithms except OPT. Figure 9(b) indicates that the average response time is reduced by up to 67% compared with DL2 and up to 22% compared with DL1.

F. Impact of consistency control

The study on the impact of consistency control on the algorithm is also carried out on the HP trace and the LLNL trace. Since HP trace is essentially metadata read dominant, its result is not as representative as LLNL trace. As the space is limited, here we only show the average response time comparison results collected on the LLNL trace, as in Figure 10. These results indicate that the average response time was

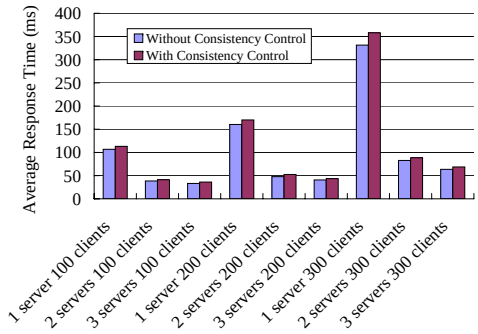


Fig. 10. Impact of consistency control

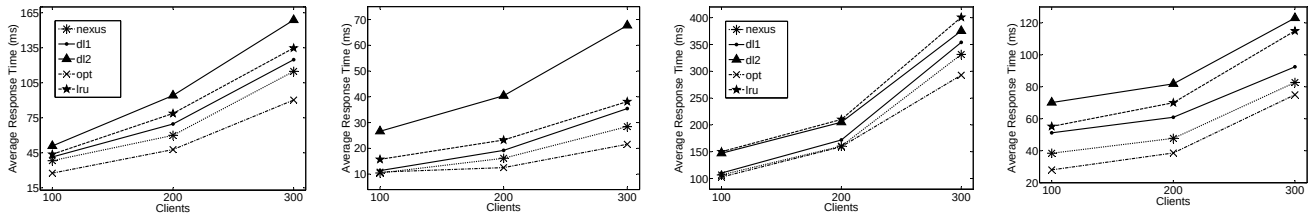
not noticeably affected by the consistency control. It shows that Nexus is not very sensitive to the write workload. A possible explanation is the characteristic of the workloads in which most of the metadata are either read-only or write-once in this scientific application.

V. Conclusions

We introduced Nexus, a novel weighted-graph-based prefetching algorithm specifically designed for clustered metadata servers. Aiming at the emerging MDS-cluster-based storage system architecture and exploiting the characteristic of metadata access, our prefetching algorithm distinguishes itself in the following aspects.

- Nexus exploits the ability to look ahead farther than the immediate successor to make wiser predictions. Sensitivity study shows that the best performance gain is achieved when the looking ahead window size is set to 5.

[2]Please note that OPT is the theoretical hit rate upper bound for pure caching approaches, not for caching/prefetching approaches.



(a) HP trace with 1 Server (b) HP trace with 2 Servers (c) LLNL trace with 1 Server (d) LLNL trace with 2 Servers

Fig. 9. Average response time comparison

- Based on the wiser prediction decision, aggressive prefetching is adopted in our Nexus prefetching algorithm to take advantage of the relatively small metadata size. Our study shows that prefetching 2 as a group upon each cache miss is optimal under the two particular traces studied. Conservative prefetching lose the chance to maximize the advantage of prefetching, and too aggressive but not so accurate prefetching might hurt the overall performance by introducing extra burden to the disk and polluting the cache.
- The relationship strengths of the successors are differentiated in our relationship graph by assigning variant edge weights. Four approaches for edge weight assignment were studied in our sensitivity study. The results show that the linear decremental assignment approach represents the most accurate strength for the relationships.
- In addition to server-oriented grouping, we also explored client-oriented grouping as a way to capture better metadata access locality by differentiating between the sources of the metadata requests. Sensitivity study results show the latter approach's consistent performance gain over the former approach, confirming our assumption.

Other than focusing on the prefetching accuracy — an indirect performance measurement, we pay our attentions to the more direct performance goal — cache hit rate improvement and average response time reduction. Simulation results show remarkable performance gains on both hit rate and average response time over conventional and state of the art caching/prefetching algorithms.

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] 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.

[6] 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.

[7] M. Gupta and M. Ammar, "A novel multicast scheduling scheme for multimedia servers with variable access patterns," Dec. 26 2002.

[8] A. Muthitacharoen, B. Chen, and D. Mazières, "A low-bandwidth network file system," in *SOSP*, 2001, pp. 174–187.

[9] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller, "Dynamic metadata management for petabyte-scale file systems," in *SC'2004 Conference CD*. Pittsburgh, PA: IEEE/ACM SIGARCH, Nov. 2004, uCSC.

[10] D. Roselli, J. R. Lorch, and T. E. Anderson, "A comparison of file system workloads," in *Proceedings of the 2000 USENIX Annual Technical Conference (USENIX-00)*. Berkeley, CA: USENIX Ass., June 18–23 2000, pp. 41–54.

[11] P. G. Sikalinda, P. S. Kritzinger, and L. O. Walters, "Analyzing storage system workloads," Jan. 01 2005. [Online]. Available: http://pubs.cs.uct.ac.za/archive/00000218/

[12] H. Lei and D. Duchamp, "An analytical approach to file prefetching," in *1997 USENIX Annual Technical Conference*, Anaheim, California, USA, Jan. 1997, http://www.cs.columbia.edu/~lei/resume.html# publications.

[13] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed prefetching and caching," in *High Performance Mass Storage and Parallel I/O: Technologies and Applications*. New York, NY: IEEE Computer Society Press and Wiley, 2001, ch. 16, pp. 224–244.

[14] J. Skeppstedt and M. Dubois, "Compiler controlled prefetching for multiprocessors using low-overhead traps and prefetch engines," *J. Parallel Distrib. Comput*, vol. 60, no. 5, pp. 585–615, 2000.

[15] A. Amer and D. D. E. Long, "Noah: Low-cost file access prediction through pairs," in *Proceedings of the 20th IEEE International Performance, Computing and Communications Conference (IPCCC '01)*. IEEE, April 2001, pp. 27–33.

[16] J. Wang, R. Min, Y. Zhu, and Y. Hu, "UCFS - a user-space, high performance, customized file system for web proxy servers," *IEEE Transactions on Computers*, vol. 51, no. 9, pp. 1056–1073, Sep 2002.

[17] A. Amer, D. D. E. Long, and R. C. Burns, "Group-based management of distributed file caches," in *ICDCS*, 2002, p. 525.

[18] [Online]. Available: http://www.base.com/gordoni/ufs93.html

[19] J. Griffioen and R. Appleton, "Reducing file system latency using a predictive approach," in *USENIX Summer*, 1994, pp. 197–207.

[20] T. T. Mclarty, "File system traces for scientific applications on large linux cluster at lawrence livermore national laboratory," Sep 2003, uCRL-MI-200102.

[21] E. Riedel, M. Kallahalla, and R. Swaminathan, "A framework for evaluating storage system security," in *FAST*, 2002, pp. 15–30.

[22] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson, "Cooperative caching: Using remote client memory to improve file system performance," University of California, Berkeley, Tech. Rep. UCB:CSD–94–844, Dec. 1994.

[23] J. Archibald and J. L. Baer, "Cache coherence protocols: Evaluation using a multiprocessor simulation model," *ACM Transactions on Computer Systems*, vol. 4, no. 4, Nov. 1986.

[24] J. S. Bucy and G. R. Ganger, "The disksim simulation environment version 3.0 reference manual," Carnegie Mellon University, School of Computer Science, Tech. Rep. CMU-CS-03-102, Jan. 2003.

[25] D. E. Knuth, "An analysis of optimal caching," *Journal of Algorithms*, vol. 6, pp. 181–199, 1985.