A Novel Weighted-Graph-Based Grouping Algorithm for Metadata Prefetching

Peng Gu*, Jun Wang†, Yifeng Zhu‡, Hong Jiang§, Pengju Shang†

*Microsoft Corporation, Redmond, WA 98052
Email:{Peng.Gu}@microsoft.com
†University of Central Florida, Orlando, Florida 32826
Email:{jwang,shang}@cs.ucf.edu
‡University of Maine, Orono, ME 04469
Email: zhu@eece.maine.edu
§University of Nebraska-Lincoln, Lincoln, Nebraska 68588
Email:jiang@cse.unl.edu

Abstract— **Although data prefetching algorithms have been extensively studied for years, there is no counterpart research done for metadata access performance. Existing data prefetching algorithms, either lack of emphasis on group prefetching, or bearing a high level of computational complexity, do not work well with metadata prefetching cases. Therefore, an efficient, accurate and distributed metadata-oriented prefetching scheme is critical to leverage the overall performance in large distributed storage systems. In this paper, we present a novel weighted-graph-based prefetching technique, built on both direct and indirect successor relationship, to reap 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 metadata prefetching algorithm, the miss rate for metadata accesses on the client site can be effectively reduced, while the average response time of metadata operations can be dramatically cut by up to 67%, compared with legacy LRU caching algorithm and existing state of the art prefetching algorithms.**

Index terms: Prefetch, algorithm, metadata, storage

I. Introduction

A novel decoupled storage architecture diverting actual file data flows away from metadata traffic 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, respectively, as shown in Figure 1.

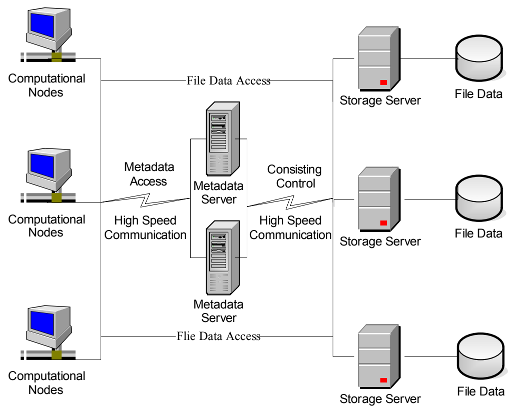


Fig. 1. System architecture

Accordingly, large 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], [6], caching [7], scheduling [8] and networking [9]. Only recent years have seen growing activities in studying the scalability of the metadata management [2], [10]–[12]. 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 for various data intensive supercomputing applications, such as predicting and modeling the effects of earthquakes and web search without language barriers. In these applications the volume of data reaches and even exceeds Peta bytes (10^{15} bytes) while metadata amounts to Tera bytes (10^{12} bytes) or more [13]. In fact, more than 50% of all I/O operations are to metadata [14], 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 particularly for metadata accesses 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, existing caching and prefetching algorithms may not work well for metadata since most caching and prefetching schemes are designed for and tested on actual file data and simply ignore metadata characteristics. As a result of this negligence, traditional caching and prefetching algorithms are not specifically optimized for metadata. And thus they may consequently not fit well with metadata access cases because file data and metadata operations usually have different characteristics and exhibit different access behaviors. For example, a file might be read multiple times while its metadata is only accessed once. An "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 (regarding this point, we will show further elucidation in section III). 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. They typically prefetch at most one file upon each cache miss. Moreover, even when a cache miss happens, certain rigid policies are enforced before issuing a prefetching operation in order to maintain a high level of prefetching accuracy. The bottom line is, they did not realize that considering the huge number and the relatively small size of metadata items, aggressive prefetching can be profitable.

On the other hand, aggressive prefetching or group-based prefetching can easily balance out their advantages by introducing 1) extra burden to the disk, 2) cache pollution and 3) high CPU runtime overhead. Hence, part of the challenges in developing an aggressive prefetching algorithm is to address the three problems at the same time.

In this paper, we make the following contributions.

- We develop a novel weighted-group-based prefetching algorithm named **Nexus** particularly for metadata accesses, featured in aggressive prefetching while maintaining adequate prefetching accuracy and polynomial runtime overhead. Although there exist group prefetching algorithms for data, the different size distributions and access characteristics between data and metadata are significant enough to justify a dedicated design for metadata access performance.
- We deploy both direct and indirect successors to better capture access localities and to scrutinize the real successor relationship among interleaved accesses sequence. Hence, Nexus is able to perform aggressive group-based prefetching without compromising accuracy. As a comparison, existing group based prefetching algorithms only consider the immediate successor relationships when building their access graphs. In other words, existing group based prefetching algorithms seem to be "short sighted" when compared with Nexus and thus potentially bear less accuracy.
- Finally, in Nexus we defined a relationship strength to build the access relationship graph for group prefetching. The way we obtain this relationship strength makes Nexus a polynomial time complexity algorithm. Other group-based prefetching algorithms, if adopted and made suitable to achieve the same level of "far sight" as Nexus does, could easily be mired in an exponential computational complexity. Therefore, Nexus distinguishes itself from others by its much lower runtime overhead.

The outline of the rest of the paper is as follows: related work is discussed in Section II. Section III shows the fundamental difference between data and metadata size distribution. Section IV describes our Nexus algorithm in detail. Evaluation methodologies and results are discussed in section V. We conclude this paper in section VI.

II. Related Work

Prefetching and caching has long been studied and implemented in modern file systems. In the area of disk level and file level prefetching, most previous work was done in three major areas: predictive prefetching [15], [16], application controlled prefetching [17]–[19], and compiler directed I/O [20], [21]. The latter two have limited applicability due to their constraints. For example, application controlled prefetching requires source code revision, and compiler directed I/O relies on a sufficient time interval between prefetching instructions inserted by the compiler and the following actual I/O instructions. Since predictive prefetching, using the past access pattern to predict future accesses, is completely transparent to clients, it is more suitable for general practice, including metadata prefetching.

Unfortunately, although the split data-metadata storage system has become ever popular for providing large scale storage solutions, there is a general negligence on the study of prefetching algorithms specifically for metadata servers: current predictive prefetching algorithms are for data but not metadata.

In order to better illustrate the difference between our Nexus algorithm and other predictive prefetching algorithms, next we briefly introduce some background in this field.

On prefetching objects in object-oriented database, Curewitz developed a probabilistic approach [22]. On prefetching whole files, Griffioen and Appleton introduced a probability graph based approach to study file access patterns [23]. In addition, Duchamp *et al.* studied an access tree based prediction approach [16]. However, in all abovementioned studies, only the immediate successors relationship are taken into consideration, while indirect successors relationship are ignored. The advantages of approaches considering both immediate and subsequent successor relationships are discussed in detail in section IV.

Based on the previous research, Long *et al.* developed a serial of successor-based predictive prefetching algorithms in their efforts to advance the prefetching accuracy while maintaining a reasonable performance gain [24]–[26]. The features of these predictors are summarized below as they are state of the art and are most relevant to our design.

a) First Successor [25]: The file that followed file A the first time A was accessed is always predicted to follow A.

b) Last Successor [25]: The file that followed file A the last time A was accessed is predicted to follow A.

c) Noah (Stable Successor) [25]: 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) [27]: 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 [27]: 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 with a predefined size *S* by following steps:

1) Add the missed item to 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 *S* is met.

Among the aforementioned five predictors, the three former ones fall into the category of single-successor predictors. If the two latter predictors are revised to take additional indirect successors into consideration for relationship graph construction, they would inevitably introduce exponential time overhead. The detail is explained in IV-D.2.

III. File data and metadata size distribution

A. Obtaining file data size distribution

To find out the difference between file data and metadata size distribution, we studied the files stored on the Franklin supercomputer [28]. Franklin is a massively parallel processing (MPP) system with 9,660 compute nodes, serving more than 1,000 users at National Energy Research Scientific Computing Center. The collection of file size distribution is somewhat straightforward compared with the metadata size case. We simply run a “ls -lR /” on the head node and then use a script to filter out the file size information from the output. Note that since we do not have the privilege to access all the files and directories stored on the system, by running these scripts we only get the size information of those files and directories that are accessible. In this study, we collected the size information for 8,209,710 regular files and 612,248 directories. The cumulative distribution function (CDF) of collected file size distribution results is shown in Figure 2.

B. Obtaining metadata size distribution

Obtaining the metadata size information is not simple. To the best of our knowledge, there is no direct way/utility in existence to find out the metadata size information for files and directories. However, there does exist a way of figuring out the corresponding metadata size if we know the file size (assuming file system type and the block size are given). For example, in an Ext2 file system, the metadata of a regular file consists of two components: a mandatory inode block and conditional indirect addressing blocks. According to latest Linux kernel source code as of this writing (version 2.6.25.9 released on June 24, 2008 [29]), each Ext2 inode structure is 128 bytes in length. This inode structure contains 12 direct block pointers plus 1 indirect block pointer, 1 double indirect block pointer and 1 triple indirect block pointer [30]. Once the block size is given, we are able to calculate the on-disk space occupied by indirect addressing blocks for files of any given size. The resulting metadata size is then the sum of the inode block size and the space for indirect addressing blocks. The detailed size mapping information between file data and metadata is summarized in Table I.

TABLE I
Size conversion between file data and metadata

Block size	Addressing Mode	File size	Metadata size
1024	Direct	$\leq$ 12 KB	128 B
	1-Indirect	12 KB$\sim$268 KB	1152 B
	2-Indirect	268 KB$\sim$64.26 MB	3200 B
	3-Indirect	64.26 MB$\sim$16.06 GB	6272 B
2048	Direct	$\leq$24 KB	128 B
	1-Indirect	24 KB$\sim$1.02 MB	2176 B
	2-Indirect	1.02 MB$\sim$513.02 MB	6272 B
	3-Indirect	513.02 MB$\sim$256.5 GB	12416 B
4096	Direct	$\leq$48 KB	128 B
	1-Indirect	48 KB$\sim$4.04 MB	**4224 B**
	2-Indirect	4.04 MB$\sim$4 GB	12416 B
	3-Indirect	4 GB$\sim$4 TB	24704 B

Note that Franklin's user home directory uses the Lustre file system, which subsequently uses Ext3 file sytem as its back-end file system [31]. Furthermore, Ext3 file system's data structures on disk are essentially identical to those of an Ext2 file system, except that it employs a journal to log metadata and data changes. This means the method we just described can be applied to calculate the metadata size based on the file data size collected. For example, given a block size of 4 KBytes (which is the basic block size for back-end Ext3 file systems chosen by Lustre file system developers) and a file size of 3 MB, the corresponding metadata size will be 4224 bytes (highlighted in Table I). Note that although this calculation applies only to Ext2 or Ext3 local file system, similar calculations can be applied to and similar conclusions can be drawn for other parallel or distributed file systems such as GPFS [32], PVFS, Panasas [33], and Ceph. All of these file systems use some form of pointers to refer to certain chunk(s) of data, regardless of whether these data are stored as regular blocks, local files, or objects.

According to the file size distribution, we obtain the corresponding metadata size distribution for the files, and the results are shown together with the file size distribution in Figure 2 for ease of comparison.

C. Directory size distribution

Metadata include both file inodes and directories: we have so far discussed the metadata size for file inodes, It may also be interesting to find out the directory size distribution. Directories are organized the same way as regular files in a Linux-based system. By *directory size* we simply mean the space in bytes occupied by those file names under the

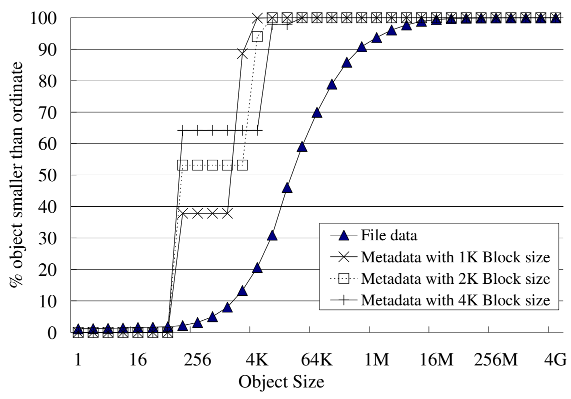


Fig. 2. Size distribution comparison of file data and metadata

directory. To obtain directory size for certain directory, we iterate all the files and sub-directories under that directory and sum the length of all the file names and sub-directory names[1]. The corresponding results are shown in Figure 3. According to these results, around 95% of the directory sizes are less than 600 bytes.

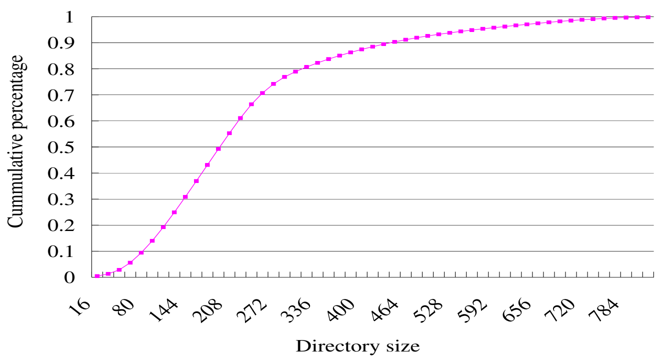


Fig. 3. Directory size distribution

D. Comparison

From the study of distinction between data and metadata size distribution on Franklin supercomputer, we observe that the file size distribution and metadata size distribution are quite different. For both data and metadata that are less than 64 bytes, the percentage is very small, i.e. less than 2%. However, around 71% of files are larger than 8 KBytes; while more than 97% of metadata are smaller than 8 KBytes under all three different block sizes. Moreover, Figure 4 shows the exact size distribution of the metadata under different block sizes in a more direct and conspicuous way.

Specifically, Figure 4(a) shows that for 1 KBytes block size, 89% of metadata are less than or equal to 1152 bytes. Figure 4(b) shows that for 2 KB block size, 94% of metadata are 2176 bytes or less. Figure 4(c) shows that for 4 KB block size, the percentage of metadata sizes larger than 4224 bytes is almost negligible.

[1]the length of file name including a ending '\0' should be rounded/aligned to a multiple of four bytes, which is an optimization done in the Ext2 file system implementation.

Based on our file data and metadata size distribution research, we observe that compared with typical file size, metadata are relatively small. We envision that the same conclusion holds for petabyte scale storage system if there is no significant change on the way the file systems manage their data and metadata. Consequently, in order to achieve optimal performance, a new prefetching algorithm that considers the size differences between data and metadata is clearly desirable. And a good example to be considered is an aggressive prefetching scheme.

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

As a more effective way for metadata prefetching, our Nexus algorithm distinguishes itself in three 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 subsequent successors. Second, Nexus exploits the fact that metadata usually is small in size and deploy an aggressive prefetching strategy. Third, Nexus maintains a polynomial runtime overhead.

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 metadatum 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. To illustrate this design, Figure 5 shows an artificially simplified example of relationship graph consisting of metadata for six files/directories. An observation obtained on this toy example is that the predecessor-successor relationship between $/usr$ and $/usr/bin$ is much stronger than that between $/usr$ and $/usr/src$.

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 MetaData Server (**MDS**) receives and serves requests from a large number of clients. A look-ahead 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 history window. Upon the arrival of a new request, the oldest request in this history window is replaced by the newcomer. 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

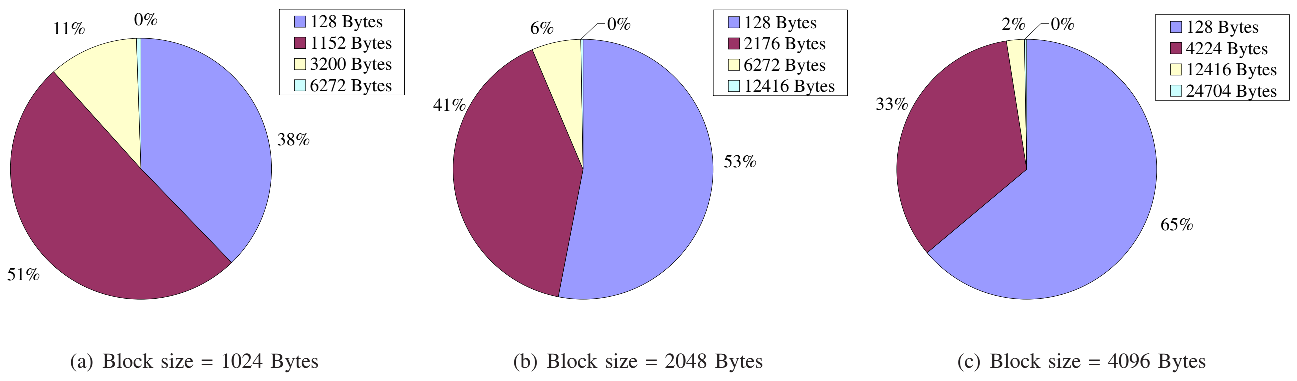


Fig. 4. Metadata size distribution

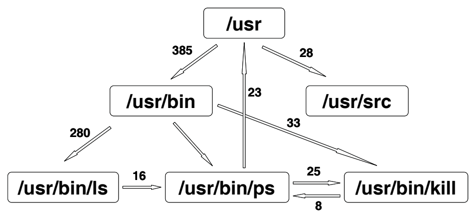


Fig. 5. Relationship graph demo

```
// Let G denote the graph to be built
BUILD-RELATIONSHIP-GRAPH($G$)
1   $G \leftarrow \varnothing$
2   for each new incoming metadata request $j$
3       for each metadata request $i$ $(i \neq j)$ in history window
4           if edge $(i, j) \notin G$
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 $j$
```

Fig. 6. Nexus grouping algorithm pseudocode

first time) or add appropriate weight to an existing edge (if this relationship has been observed before).

A piece of pseudocode describing how the relationship graph is built from the beginning is provided in Figure 6 and an example is given in Figure 7 for better understanding.

In this example, an sample request sequence of

$$ABCADCBA\cdots$$

is given. Figure 7(a) shows the step by step graph construction from scratch with a history window size of two (The weight assignment methodology assumed here is linear decremental, described later in Section IV-E.1 on page 7). In contrast, Figure 7(b) shows the same relationship graph construction procedure with a history window size of three.

C. Prefetching based on the relationship graph

Once the graph is built for the access sequence $ABCADCBA\cdots$ as shown in Figure 7(a) or Figure 7(b), we are now ready to prefetch a number of successors as a group with a configurable size in the graph when a cache miss happens for an element in that group. The prediction result depends on the order of the weights (represented by numbers associated with arrows in Figure 7) of outbound edges originated from the latest missed element. A larger weight indicates a closer relationship and a higher prefetching priority. Assuming the last request A in the above sample access sequence sees a miss, according to the graph shown in Figure 7(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 7(b) will be $\{B\}$ and $\{B, C\}$, respectively (as shown in Table II).

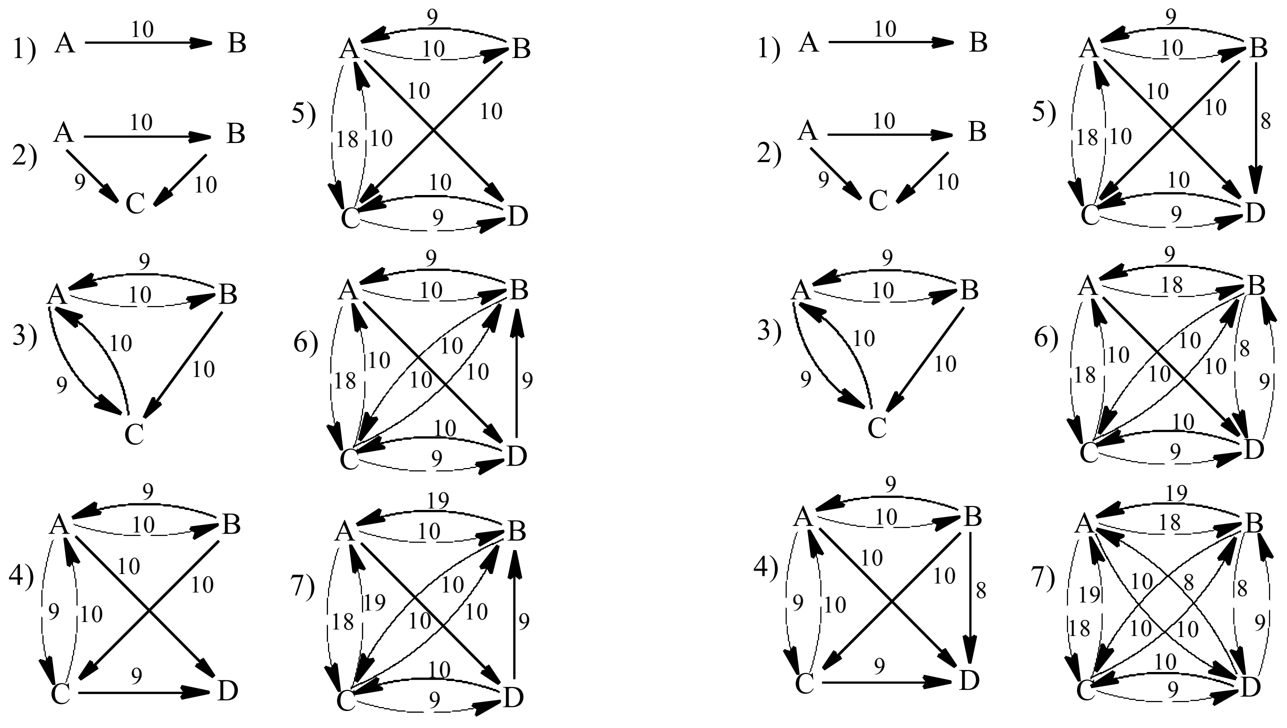
TABLE II
PREDICTION RESULTS COMPARISON. P1 MEANS PREFETCHING WITH GROUP SIZE = 1; P2 MEANS PREFETCHING WITH GROUP SIZE = 2; N2 MEANS NEXUS WITH HISTORY WINDOW SIZE = 2; N3 MEANS NEXUS WITH HISTORY WINDOW SIZE = 3

	N2	N3
P1	C	B
P2	CD	CB

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



(a) Look-ahead history window size = 2 (b) Look-ahead history window size = 3

Fig. 7. Graph construction examples

played is loaded; at last the decoder program ("B") for that type of media is loaded.

In addition to above mentioned applications, interleaved application I/Os coming from multi-core computers or from many clients will only make things worse. The probability-based model can not detect such access patterns, thus limiting its ability to make better predictions. However, this omitted information is taken into consideration in our relationship-based prefetching algorithm, which is able to look farther than the immediate successor when we build our relationship graph.

We use the same aforementioned sample trace sequence, $ABCADCBA\cdots$, 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 *indirect* 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 thus we predict this pattern will repeat in the near future. As discussed in IV-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.

2) Farther sight within small overhead: The aforementioned advantage comes at the cost of a look-ahead history window. This approach appears to be prohibitive for other online prefetching algorithms due to potential high runtime overhead. However, this overhead is kept minimum in our design. In fact, we actually achieved a polynomial time complexity for our relationship graph construction algorithm as shown in Figure 6.

Theorem 1: The Nexus grouping algorithm given in Figure 6 bears polynomial time complexity

Proof: Let L denote the look-ahead history window size; let n denote the length of the entire metadata access history. We will first calculate the time required by each step described in Figure 6 and then derive the aggregated algorithm complexity. Step 1 always takes constant time, i.e., $O(1)$. Step 2 dictates that step 3 through 7 should be executed n times. Consequently steps 3 dictates that steps 4 through 6 should run L times. Step 4 requires constant time assuming a two-dimensional adjacency matrix representation is adopted for graph G. Either step 5 or step 6 is chosen to be executed next according to runtime conditions. Step 5 requires $O(N^2)$ and step 6 requires $O(1)$ constant time, regardless of which one is selected, the worst case scenario is $O(N^2)$ for step 5 and 6 combined. Step 7 also takes constant time, as it replaces the oldest item by overwriting the array element in the circular history window pointed by the last-element-pointer and shifting that pointer to the next element, thus no scanning or searching is involved. Putting it all together, the worst case time complexity for this algorithm is $O(1)+O(n)\cdot\{O(L)\cdot[O(1)+O(N^2)]+O(1)\} = O(n^3\cdot L)$, which means a polynomial time complexity. ■

In contrast, should we apply the same idea to a probability-based approach, the complexity of the algorithm would be exponential. For example, if look-ahead history window size is set to 2 (i.e., L=2) rather than 1 (L=1 means only looking at the immediate successor), a probability-based approach would maintain the conditional probability per 3-tuple $P(C|AB)$ instead of per 2-tuple $P(B|A)$. Under the same assumption for graph representation as used in the proof above, we can prove that the time complexity will be $O(n^L)$ for probability-based approach as opposed to $O(n{\cdot}L)$ for Nexus. If we choose to switch to adjacency list graph representation for the sake of potential less memory usage[2], the algorithm time complexity would grow to prohibitive $O(n^{L+2})$ for a probability-based approach while only $O(n^3 \cdot L)$ for Nexus.

3) 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 [34]. 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 is the observation that 99% of metadata are less than 4224 bytes, while 40% of file data are larger than 4 KB, as observed in III-D. On the other hand, we also observe that metadata servers and compute nodes equipped with multiple gigabytes or even terabytes of memory become norm. These observations encourage us to conduct aggressive prefetching on metadata, considering that a single cache miss at the client site will result in a mandatary network round-trip latency plus potential disk operation overhead when the requested metadata server consequently sees a cache miss.

E. Algorithm design considerations

When 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,

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

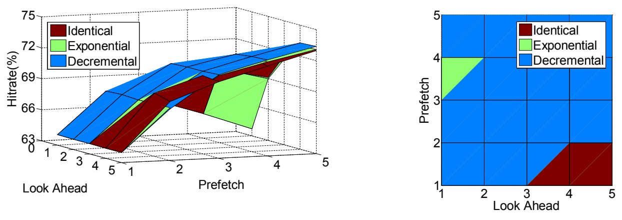
respectively. Four approaches are taken into consideration:

- *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 [23]. It may look simple and straightforward, but it is indeed effective. The key point is that at least the successors following the immediate successors are taken into consideration. However, the drawback 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 7(a) and Figure 7(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 an increase in the successor distance, the decrease in the relationship strength might be more radical than the linear one. For example, polynomial decremental assignment is a possible alternative solution. This assumption is based on the observation of the attenuation of radiation in the air in real life.
- *Exponential decremental assignment* The attenuation of edge weights might be even faster than polynomial decremental. In this case, an exponential decrement model is adopted. This approach is referred to as *exponential* decremental assignment in the future.

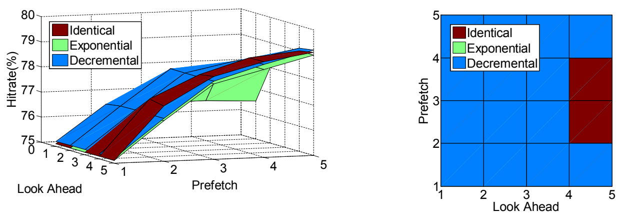
To find out which assignment method can best reflect the locality strength in the metadata reference streams, we conducted experiments on the HP file server trace [14] to compare the hit rate achieved by those four edge-weight assignment methods. To be comprehensive, these experiments are conducted with 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). In our experiments, the cache size (as a fraction of total metadata workset size) varies from 10% to 90% in an ascending step of 20%. We found that the effects of prefetching become negligible once the cache size exceeds 50%. Accordingly, in this paper, we only present the results with cache size of 10%, 30% and 50%. In addition, we also observe that the results for the polynomial assignment is very close to those for the exponential assignment, so we remove the former results to show readers a clearer figure. The results for the remaining three approaches are shown in Figure 8.

In Figure 8, the 3D graphs on the left show the hit rate achieved by those three approaches over three different cache size configurations (i.e. 10%, 30% and 50%) with both the look-ahead history window size and prefetching group size varying from 1 to 5. (The 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 in some cases even though this approach is very simple.

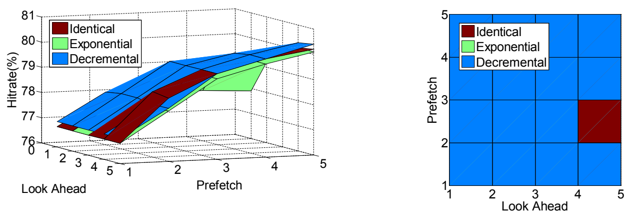
[2]Switching to adjacency list representation may reduce memory space occupation at the cost of potential computing time increase if the original adjacency matrix turns out to be a sparse matrix.



(a) Cache size = 10%



(b) Cache size = 30%



(c) Cache size = 50%

Fig. 8. Edge weight assignment approaches comparison

Since the linear decremental assignment approach consistently outperforms others, 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 noise 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 history window size and prefetching group size. In these experiments we adopt the same simulation framework described in Section V-B. The result is shown in Figure 9. From Figure 9, 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 look-ahead history window size, the better the hit rate achieved. This observation prompts us to experiment on much larger look-ahead history window, with sizes 10, 50, and 100 respectively, and found contradicting results to our conjecture: none of those three look-ahead history 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 effectiveness of the relationships captured by the look-ahead history window.

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. In addition, since a cache size as small as 10% is good enough to demonstrate this performance gain, we will use this as the default configuration unless otherwise specified.

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 client nodes 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 on the metadata servers: 1) Build a single relationship graph for all the requests received by a particular metadata server; or 2) Build a relationship graph for requests originated 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 10.

Figure 10 clearly shows that client-oriented grouping algorithm consistently outperforms the server-oriented one. Thus we adopt the client-oriented grouping algorithm whenever possible.

4) Weights overflow issue: As the edge weights in the relationship graph are monotonically non-decreasing, over time integer overflow is going to happen sooner or later. One possible solution to this problem is using some forms of "big Integer" library that represents integers of arbitrary size. For example, here is one such library [35]. However, those libraries may introduce some unexpected overhead due to the increasing size of data.

Another way to address overflow problem is to re-calculate all the weights when one or more of them are going to be overflowed. Our purpose is to avoid integer overflow while keeping the order of the weights (do not want to change the priorities of the vertices connected to this vertex). An example shown in Figure 11 illustrates this idea.

In Figure 11, weight of edge (i,j) means the quantified relationship from vertex i to j. If data overflow is detected when edge (i, j) is being renewed, the weights of all the edges starting from vertex i are simply re-calculated as shown in the right part of Figure 11. Since it is easy to enumerate the edges originated from vertex i (no matter adjacent matrix or list is used), we can re-assign new weights from 0 to N-1 (assume N is the number of edges whose start point is vertex i) to the edges while make sure the original weight ordering is not changed. In Figure 11, there are 5 edges $E1 \sim E5$ in original graph, the order is $E1 > E2 > E5 > E3 > E4$,

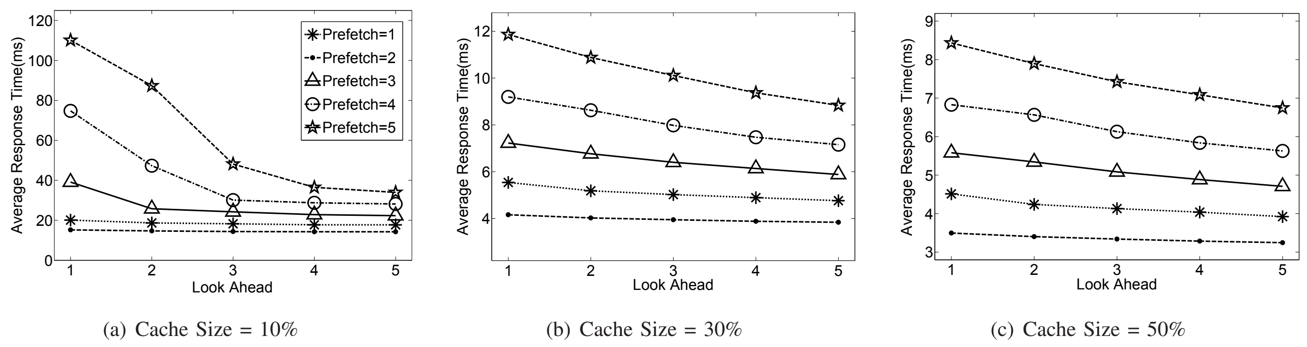


Fig. 9. Sensitivity Study: Look Ahead and Prefetch

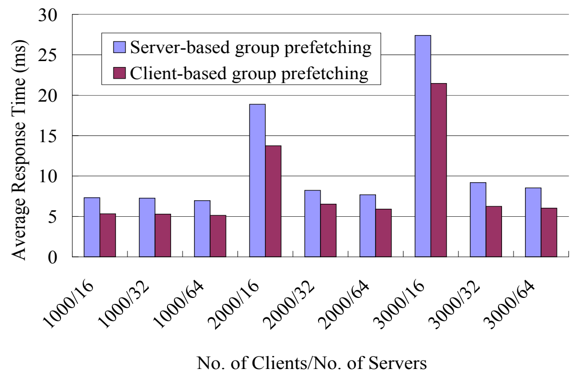


Fig. 10. Server-Oriented Grouping VS Client-Oriented Grouping

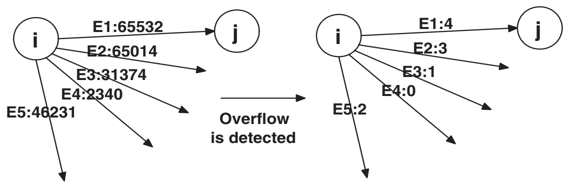


Fig. 11. One way to solve the integer overflow problem

which is also kept in the re-calculated graph. Although the new weights are way less stronger than the old ones (for example, in the left part of Figure 11, even if the weights of E5 is added multiply times, it still can not exceed the priority of E2. While in the right part of Figure 11, a single addition to the weight of E5 will change the ordering of prefetching), they can still guarantee the right order for the immediate succeeding prefetching. In other words, we reset the priorities of prefetching candidates for metadata i to solve the integer overflow problem, by partially sacrificing the formerly accumulated accuracy of weights order. The time complexity of this operation is O(nlog(n)), which is the optimized sorting time.

For the sake of simplicity, normal integers (without overflow) are used in the proof and experiments.

V. Evaluation Methodology and Results

This section describes the workload, the simulation framework, and the detailed simulation we used to evaluate the metadata performance equipped with Nexus. The metrics we used here include hit rate and average response time. In addition, we also studied the impact of consistency control and scalability of Nexus algorithm.

TABLE III
List of operations obtained by *strace* in LLNL trace collection

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

A. Workloads

We evaluate our design by running trace-driven simulations over one scientific computing trace and one file server trace: the LLNL trace collected at Lawrence Livermore National Laboratory in July 2003 [36] and the HP-UX server file system trace collected at the University of California Berkeley in December 2000 [37]. 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 LLNL 2003 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. Since the LLNL trace is collected at the file system level, any requests not related to metadata operations, such as read, write and execution, are filtered out. Table III manifests the remaining metadata operations in the LLNL trace. These metadata operations are further classified into two categories: metadata read and metadata write before fed into the simulations discussed in Section V-B and Section V-C. 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 $close$ is not straight forward. 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 or may not 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 $close$, an eclectic way is that 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 [37], 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 these traces are relatively old, we scale up the workload collected in this environment to better emulate the projected more intensive workload in a petabyte storage system. We divide each daily trace collected from 8:00am to 16:00pm, which were usually the busiest period during a day, into four fragments, with each fragment containing two hours of I/O accesses. The time stamps of all events in each fragment are then equally shifted so that this fragment starts at time instant zero. Replaying multiple time-shifted fragments simultaneously increases the I/O arrival rate while keeping a similar histogram of file system calls. In addition, the number of files stored and the number of files actively visited were scaled up proportionally by adding the date fragment number as a prefix to all filenames. We believe that replaying a large number of processed fragments together can emulate the workload of a larger cluster without inadequately break the original access patterns at the file system level. Same as what we did for the LLNL trace, we also filtered out those metadata-irrelevant I/O operations in our simulations.

B. Simulation framework

A simulation framework was developed to simulate a clustered MDS based storage system with the ability to adopt flexible caching/prefetching algorithms. The simulated system consists of 1000 to 8000 compute nodes (clients) and 4 to 256 MDSs. The memory size is set to be 4 GB per MDS and 1 GB per client. All nodes are connected using high speed interconnection with an average network delay of 0.3 ms and a bandwidth of 1 Gbit/sec under assumption of a standard Gigabit Ethernet environment [38]. The interconnect configuration is the same as shown in Figure 1. In such a large, hierarchical, distributed storage system, metadata consistency control on metadata servers as well as the clients becomes a prominent problem for the designers. However, the focus of our current study is the design and evaluation of a novel prefetching algorithm for metadata. To simplify our simulation design, cooperative caching [39], a widely used hierarchical cache design, together with its cache coherence control mechanism, i.e. write-invalidate [40], is adopted on the metadata servers in our simulation framework to cope with the consistency issue. The specific cooperative caching algorithm we adopted is N-chance Forwarding, the most advanced solution according to the results presented in [39]. We choose the best cooperative caching solution available for the sake of fair performance comparison. This aims to evaluate the real performance gain from Nexus. From this aspect, it also helps to distinguish the effect of Nexus from that of cooperative caching.

It may also be noticed 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 our simulation framework, the storage system consists of four layers: 1) client local cache, 2) metadata server memory, 3) cooperative cache, and 4) hard disks. When the system receives a metadata request, it first checks its local 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: client local hit, metadata server memory 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 local cache, referred to as a *local hit*, 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, also known as a *server memory hit*, 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, defined as a *remote client hit*, extra network delay should be considered. In the worst case, when the MDS has to send the request to the disks where the requested metadata resides, i.e., a final cache *miss*, costly disk access overhead will also contribute to the response time.

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

C. Trace-driven simulations

Trace-driven simulations based on aforementioned HP trace and 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), primitive prefetching algorithms such as First Successor and 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 PG in the rest of this paper).

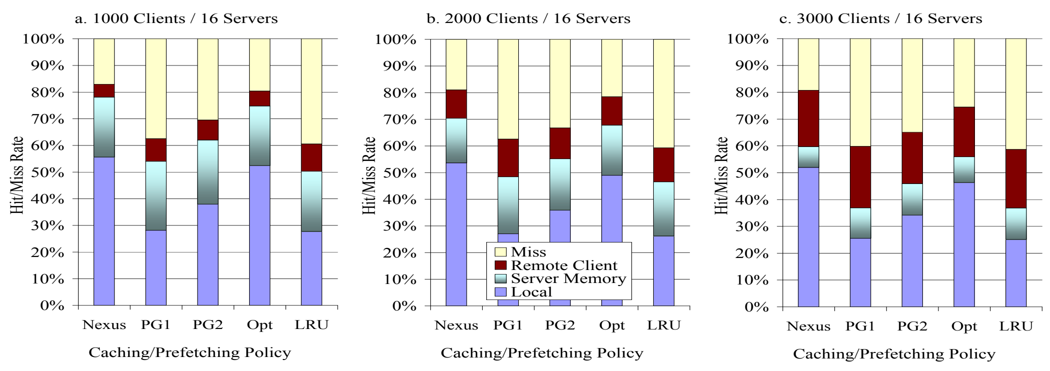


Fig. 12. HP trace hit rate comparison

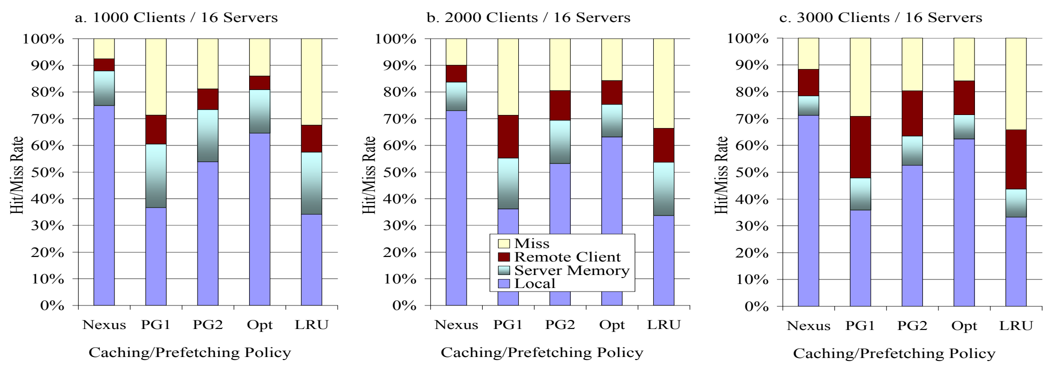


Fig. 13. LLNL trace hit rate comparison

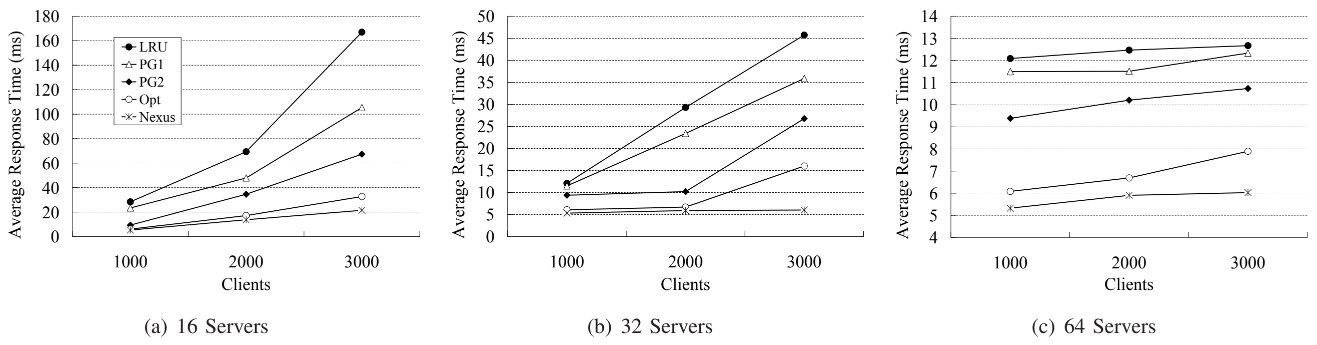


Fig. 14. Comparisons of HP Average Response Time per Metadata Request

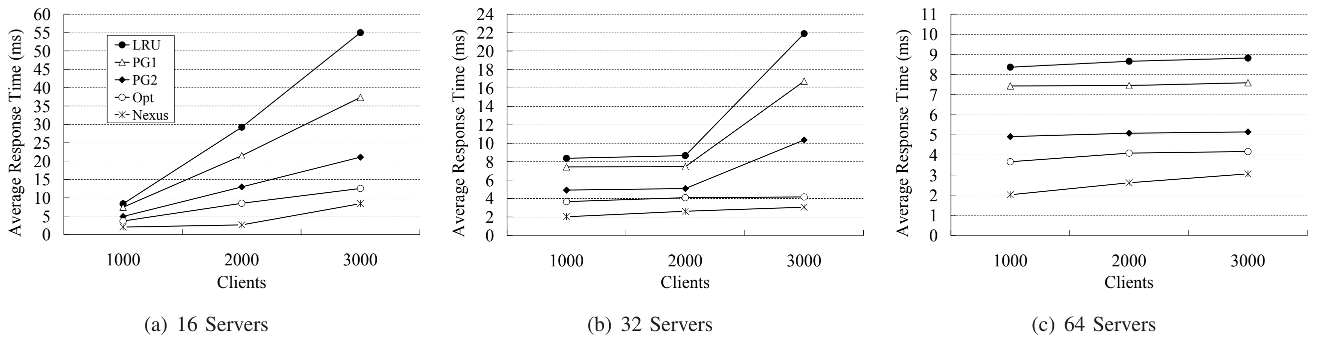


Fig. 15. Comparisons of LLNL Average Response Time per Metadata Request

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. The reason is, a conservative prefetching scheme, even with a high prefetching accuracy, might incur little prefetching actions and thus not as beneficial. So, in our experiments, we not only measure the cache hit rate, but also the average response time by integrating a golden disk simulator, DiskSim 3.0 [41], into our simulation framework.

We conduct experiments for all the caching/prefetching algorithms mentioned above. For a clear graphic presentation, we remove the results for less representative algorithms, including LFU, MRU (these two are always worse than LRU), First Successor, Last Successor, Noah, and Recent Popularity, since these algorithms are consistently inferior to PG according to our experimental results and a similar observation made by Pâris in [42]. In addition to these algorithms, Optimal Caching [43], referred to as *OPT* in the rest of this paper, is simulated as an ideal offline 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 PG, referred to as PG1 and PG2, 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, PG1, PG2, LRU and OPT.

D. Hit rate Comparison

We have collected the hit rate results for all three levels of caches: client cache, server cache and cooperative cache, as well as the percentage of misses that goes to the server disk, referring to the explanation in V-B.

Figure 12 and Figure 13 show the hit rate comparison results collected on HP trace and LLNL trace, respectively.

Comparing Figure 12(a), 12(b) and 12(c), it is apparent that with more clients, and thus larger cooperative cache size and smaller per-client server cache size, many requests previously satisfied by the server cache is now caught by the cooperative cache. However, the client local cache hit rate and the overall cache hit rate stay relatively consistent.

In both Figure 12 and Figure 13, Nexus achieves noticeable better performance on the client local cache hit rate than the other four competitors. For example, Nexus can achieve up to 40% higher local hit rate than that of LRU and PG1. In addition, the fact that PG2 obtains consistent higher client local cache hit rate than PG1 is another implication that advocates the general idea of group prefetching. Based on this reasoning, it seems that a projected PG3 algorithm may potentially outperform PG2 significantly, but its exponential computational complexity prohibited us from further exploring in this direction. It is worth reminding that, Nexus only incurs linear or polynomial computational overhead and thus suits well for group prefetching.

It is surprising to see that Nexus even beats Opt by a small margin (around $3\sim10\%$) in terms of local hit rate, given Opt being the optimal offline caching algorithm with an unrealistic advantage to actually "see" future request sequence before making cache replacement decisions. The only limitation of Opt is the lack of prefetching capability compared with Nexus. Consider the situation where object A, B, C and D are always accessed as a group but none of them are currently in the cache, Opt bears four cache misses. However, Nexus will prefetch B, C and D upon a cache miss for A, resulting in one cache miss and three hits.

It may also be worth mentioning that even though Nexus achieves the highest client local cache hit rate, its advantage on overall hit rate is somewhat offset by server cache and cooperative cache. On the other hand, this observation confirms that even the best cooperative caching scheme cannot replace Nexus. At any rate, the overall hit rate does not fully and truly show the merits of Nexus prefetching algorithm. Instead, it is the client cache hit rate that may exhibit the benefits of Nexus. More importantly, even server cache hit and cooperative cache hit come at the cost of network delay in the range of milliseconds, considerably slower than a local hit which incurs only memory access latency in the range of nanoseconds.

E. Average Response Time Comparison

Taking into consideration the possibility that the advantage of prefetching be compromised if too many extra disk accesses are introduced, to accurately measure average response time, we adopted an established disk simulator to incorporate the disk access time in our simulation. The procedure of how each single request is serviced is given detailed explanations in V-B. In the experiments, We collect the results for both HP trace and LLNL trace and present their results in Figure 14 and Figure 15, respectively.

Apparently, the Nexus algorithm excels in all cases in Figure 14(a) and Figure 15(a). With 16 servers, increasing the number of clients from 1000, 2000 to 3000 results in considerable longer average response time for all algorithms. In contrast, with 32 servers, the average response time for Nexus in Figure 14(b) and that of Nexus and Opt in Figure 15(b) stays nearly constant while others increase significantly. Furthermore, in Figure 14(c) and Figure 15(c), the average response time for all algorithms seems to stay little changed. Based on these observations, it seems that individual algorithms exhibit different degrees of "sensitivity" to increasingly intensive workloads. More specifically, systems running the Nexus or Opt algorithm are less likely to be saturated under the same workload.

The advantage of Nexus comes from two aspects. First of all, as shown in Figure 12 and Figure 13, the local hit rate and overall hit rate of Nexus are higher than the others. In addition, the computational overhead of this algorithm is kept minimal. Given these advantages, even in cases where the workload stress is relatively high (see Figure 14(a) and Figure 15(a)), Nexus shows a moderate increase in average response time, in contrast to the much more dramatic increase exhibited by other algorithms.

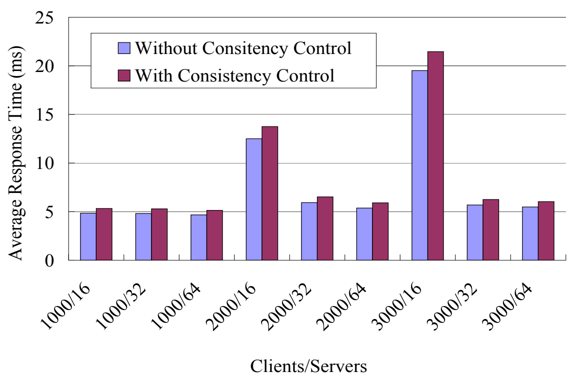


Fig. 16. Impact of consistency control

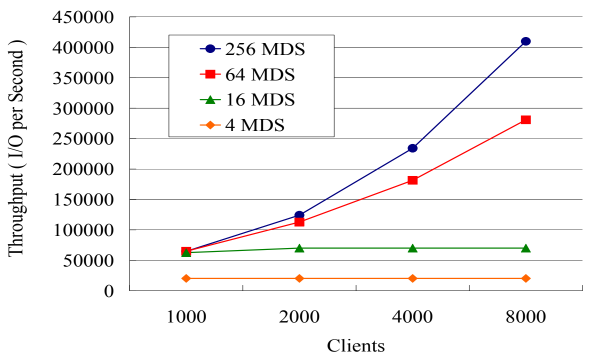


Fig. 17. Scalability study using HP trace

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. As the results for LLNL trace and HP trace are similar, here we only show the average response time comparison results collected on the HP trace, as in Figure 16.

These results indicate that the average response time was not noticeably affected by the consistency control, within a range of only 5∼10%. In other words, consistency control does not entangle Nexus very much. A possible explanation is that the characteristic of the metadata workloads in this application are either read-only or write-once. In a write intensive workload, the impact of consistency control may become more noticeable. Regarding to the applicability of Nexus in a practical system, similar to other prefetching/caching algorithms, our scheme works better for read dominant applications than write dominant applications in order to avoid excessive overhead incurred by the consistent control policy.

G. Scalability study

In a multi-client multi-MDS storage environment, the system scalability is an important factor directly related to the aggregated system performance. We studied the scalability of the metadata servers equipped with Nexus prefetching algorithm by simulating large numbers of clients and servers. Our evaluation methodology is that keeping constant number of metadata servers, we increase the number of clients and measured the corresponding system throughput, defined by the aggregate number of metadata I/Os serviced per second by the metadata servers. The results in Figure 17 show that, given 4 servers, the throughput does not significantly increase while the number of clients increase from 1000 to 8000, as the system is already saturated by 1000 clients at the first place. Prefetching simply can not help when the system is overloaded. In the 16-server case, the throughput increases approximately 6% when the number of clients increase from 1000 to 2000, after that it stops growing since the system became saturated. With 64 or 256 servers, the system throughput scales up almost proportionally with the number of clients, indicating near optimal scalability of the system. As an example, in the 256-server case, the throughput grows from about 6.5×10^4 I/O per second with 1000 clients to about 4.1×10^5 with 8000 clients, more than 6 times increase is achieved.

There are three major factors that contributes to its scalability. First, Nexus algorithm is totally distributed to clients nodes, there is no central control in our design. System scalability is given serious consideration at the time of Nexus algorithm design. Second, Nexus algorithm runs on client site. That means increased number of clients also provide additional computation power for this algorithm. Third, there is no inter-client communication involved, eradicating the most prominent factor that limits the scalability in many distributed systems.

VI. 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 look-ahead history window size is set to 5.
- 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.

VII. Acknowledgments

This work is supported in part by the US National Science Foundation under grants CNS-0646910, CNS-0646911, CCF-0621526, CCF-0811413, CCF-0621493 and the US Department of Energy Early Career Principal Investigator Award DE-FG0207ER25747.

References

[1] P. Schwan, "Lustre: Building a file system for 1000-node clusters," *Proc. 2003 Linux Symp.*, 2003.

[2] Y. Zhu, H. Jiang, and J. Wang, "Hierarchical bloom filter arrays (HBA):a novel, scalable metadata management system for large cluster-based storage," *Proc. IEEE Int'l Conf. on Cluster Computing*, pp. 165–174, Oct 2004.

[3] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," *Proc. 9th ACM Symp. on Operating systems principles*, pp. 29–43, 2003.

[4] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: a scalable, high-performance distributed file system," *Proc. 7th Conf. on USENIX Symp. on Operating Systems Design and Implementation*, pp. 22–22, 2006.

[5] I. F. Haddad, "Pvfs: A parallel virtual file system for linux clusters," *Linux J.*, p. 5, 2000.

[6] J. H. Hartman and J. K. Ousterhout, "The zebra striped network file system," *ACM Trans. Comput. Syst.*, vol. 13, no. 3, pp. 274–310, 1995.

[7] E. Otoo, D. Rotem, and A. Romosan, "Optimal file-bundle caching algorithms for data-grids," *Proc. ACM/IEEE Conf. on Supercomputing*, p. 6, 2004.

[8] M. Gupta and M. Ammar, "A novel multicast scheduling scheme for multimedia servers with variable access patterns," *Proc. IEEE Int'l Conf. on Communications*, May 2003.

[9] A. Muthitacharoen, B. Chen, and D. Mazières, "A low-bandwidth network file system," *Proc. 8th ACM Symp. on Operating systems principles*, pp. 174–187, 2001.

[10] P. Gu, Y. Zhu, H. Jiang, and J. Wang, "Nexus: A novel weighted-graph-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.

[11] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn, "Crush: controlled, scalable, decentralized placement of replicated data," *Proc. 2006 ACM/IEEE Conf. on Supercomputing*, p. 122, 2006.

[12] A. Devulapalli and P. Wyckoff, "File creation strategies in a distributed metadata file system," *Proc. 21st IEEE Int'l Parallel and Distributed Processing Symp.*, pp. 1–10, 2007.

[13] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller, "Dynamic metadata management for petabyte-scale file systems," *Proc. ACM/IEEE Conf. on Supercomputing*, pp. 4–4, nov 2004.

[14] D. Roselli, J. R. Lorch, and T. E. Anderson, "A comparison of file system workloads," *Proc. USENIX Annual Technical Conf.*, pp. 41–54, Jun. 18–23 2000.

[15] D. Kotz and C. S. Ellis, "Practical prefetching techniques for multiprocessor file systems," *J. Distributed and Parallel Databases*, vol. 1, no. 1, pp. 33–51, Jan. 1993.

[16] H. Lei and D. Duchamp, "An analytical approach to file prefetching," *Proc. USENIX Annual Technical Conf.*, Jan. 1997.

[17] A. Tomkins, "Practical and theoretical in prefetching and caching," *PhD dissertation*, 1997.

[18] R. H. Patterson, G. A. Gibson, and M. Satyanarayanan, "A status report on research in transparent informed prefetching," *ACM Operating Systems Review*, vol. 27, no. 2, pp. 21–34, 1993.

[19] P. Cao, E. W. Felten, A. Karlin, and K. Li, "Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling," *ACM Trans. Computer System*, vol. 14, no. 4, pp. 311–343, Nov. 1996.

[20] 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.

[21] T. C. Mowry, A. K. Demke, and O. Krieger, "Automatic compiler-inserted I/O prefetching for out-of-core applications," *Proc. 2nd USENIX Symp. on Operating Systems Design and Implementation*, pp. 3–17, 1996.

[22] K. M. Curewitz, P. Krishnan, and J. S. Vitter, "Practical prefetching via data compression," *Proc. ACM Int'l Conf. on Management of Data*, pp. 257–266, May 1993.

[23] J. Griffioen and R. Appleton, "Reducing file system latency using a predictive approach," *Proc. USENIX Summer 1994 Technical Conf.*, Jun. 6–10 1994.

[24] T. M. Kroeger and D. D. E. Long, "The case for efficient file access pattern modeling," in *HOTOS '99: Proceedings of the The Seventh Workshop on Hot Topics in Operating Systems*. Washington, DC, USA: IEEE Computer Society, 1999, p. 14.

[25] A. Amer and D. D. E. Long, "Noah: Low-cost file access prediction through pairs," *Proc. 20th IEEE Int'l Performance, Computing and Communications Conf.*, pp. 27–33, April 2001.

[26] T. M. Kroeger and D. D. E. Long, "Design and implementation of a predictive file prefetching algorithm," in *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2001, pp. 105–118.

[27] D. L. A. Amer and R. Burns, "Group-based management of distributed file caches," *Proc. 22nd International Conference on Distributed Computing Systems*, p. 525, 2002.

[28] NERSC, "Franklin - Cray XT4," http://www.nersc.gov/nusers/systems/franklin/, Oct 2008.

[29] "The linux kernel archives," 2007. [Online]. Available: http://www.kernel.org

[30] D. D. Bovet and M. Cesati, "Understanding the Linux kernel," *O'Reilly Associates, Inc.*, p. 923, 2005.

[31] S. Microsystems, "Lustre FAQ," http://wiki.lustre.org/index.php?title=Lustre_FAQ#Why_did_Lustre_choose_ext3.3F_Do_you_ever_plan_to_support_others.3F, April 2008.

[32] F. Schmuck and R. Haskin, "GPFS: A shared-disk file system for large computing clusters," pp. 231–244, Jan., 2002.

[33] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou, "Scalable performance of the panasas parallel file system," pp. 17–33, Feb 2008.

[34] X. Zhuang and H.-H. S. Lee, "Reducing cache pollution via dynamic data prefetch filtering," *IEEE Trans. Computers*, vol. 56, pp. 18–31, Jan 2007.

[35] "C++ Big Integer Library," http://mattmccutchen.net/bigint/.

[36] F. Wang, Q. Xin, B. Hong, S. A. Brandt, E. Miller, D. Long, and T. McLarty, "File system workload analysis for large scale scientific computing applications," Jan 2004.

[37] E. Riedel, M. Kallahalla, and R. Swaminathan, "A framework for evaluating storage system security," *Proc. 1st USENIX Conf. on File and Storage Technologies*, pp. 15–30, 2002.

[38] "IEEE 802.3 standard," 2005. [Online]. Available: http://standards.ieee.org/getieee802/802.3.html

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

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

[41] J. S. Bucy and G. R. Ganger, "The disksim simulation environment version 3.0 reference manual," Carnegie Mellon Univ., School of Computer Science, Tech. Rep., Jan. 2003.

[42] J.-F. Pâris, A. Amer, and D. D. E. Long, "A stochastic approach to file access prediction," *Proc. Int'l workshop on Storage network architecture and parallel I/Os*, pp. 36–40, 2003.

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

Peng Gu received the BS and MS degrees in computer science from the Huazhong University of Science and Technology, Wuhan, China, and the PhD degree in computer engineering from the University of Central Florida, Orlando. He is a software engineer in the Core Operating System Division, Microsoft Corp., Redmond, Washington. His research interests include file and storage systems, parallel I/O architecture, and high performance computing.

Jun Wang received the BEng degree in computer engineering from Wuhan University (formerly Wuhan Technical University of Surveying and Mapping), the MEng degree in computer engineering from the Huazhong University of Science and Technology, China, and the PhD degree in computer science and engineering from the University of Cincinnati in 2002. He is a member of the faculty of the School of Electrical Engineering and Computer Science, University of Central Florida, Orlando. His research interests include I/O architecture, file and storage systems, parallel and distributed computing, cluster and P2P computing, and performance evaluation. He has received several major US National Science Foundation research awards from Computer Systems Research program and Advanced Computation Research Program respectively, and the US Department of Energy Early Career Principal Investigator Award program. He is a member of the IEEE, the ACM and the Usenix.

Yifeng Zhu received the BSc degree in electrical engineering from the Huazhong University of Science and Technology, Wuhan, China, in 1998 and the MS and PhD degrees in computer science from the University of Nebraska, Lincoln, in 2002 and 2005, respectively. He is currently an assistant professor in the Department of Electrical and Computer Engineering, University of Maine. His research interests include parallel I/O storage systems, supercomputing, energy aware memory systems, and wireless sensor networks. He served as the program chair of IEEE NAS ’09 and SNAPI ’07, the guest editor of a special issue of the International Journal of High Performance Computing and Networking, and the program committee of various international conferences, including ICDCS, ICPP, and NAS. He received Best Paper Award at IEEE CLUSTER ’07 and several research and education grants from the US National Science Foundation HECURA, ITEST, REU, and MRI. He is a member of the ACM, the IEEE, the IEEE Computer Society, and the Francis Crowe Society.

Hong Jiang received the B.Sc. degree in Computer Engineering in 1982 from Huazhong University of Science and Technology, Wuhan, China; the M.A.Sc. degree in Computer Engineering in 1987 from the University of Toronto, Toronto, Canada; and the PhD degree in Computer Science in 1991 from the Texas A&M University, College Station, Texas, USA. Since August 1991 he has been at the University of Nebraska-Lincoln, Lincoln, Nebraska, USA, where he served as Vice Chair of the Department of Computer Science and Engineering (CSE) from 2001 to 2007 and is Professor of CSE. At UNL, he has graduated 10 Ph.D. students who upon their graduations either landed academic tenure-track positions or were employed by major US IT corporations. His present research interests include computer architecture, computer storage systems and parallel I/O, parallel/distributed computing, cluster and Grid computing, performance evaluation, real-time systems, middleware, and distributed systems for distance education. He serves as an Associate Editor of the IEEE Transactions on Parallel and Distributed Systems. He has over 150 publications in major journals and international Conferences in these areas, including IEEE-TPDS, IEEE-TC, JPDC, ISCA, FAST, ICDCS, OOPLAS, ECOOP, ICS, HPDC, ICPP, etc., and his research has been supported by NSF, DOD and the State of Nebraska. Dr. Jiang is a Member of ACM, the IEEE Computer Society, and the ACM SIGARCH.

Pengju Shang received his BS degree from Jilin University, Changchun, China, MS degree from Huazhong University of Science and Technology, Wuhan, China. Now he is a PhD student in computer engineering school of University of Central Florida, Orlando. His specific interests include transaction processing, RAID systems, Storage architecture, and Neural Networks.