Prefetching on Storage Servers through Mining Access Patterns on Blocks

Jianwei Liao, François Trahay, Balazs Gerofi, and Yutaka Ishikawa, *Member, IEEE*

Abstract—Distributed file systems have been widely deployed as back-end storage systems to offer I/O services for parallel/distributed applications that process large amounts of data. Data prefetching in distributed file systems is a well-known optimization technique which can mask both network and disk latency and consequently boost I/O performance. Traditionally, data prefetching is initiated by the client file systems, however, conventional prefetching schemes are not well suited for client machines that have limited memory and computing capacity. To offer an efficient prefetching approach for resource-limited client machines, this paper proposes a novel server-side prefetching mechanism. Specifically, we propose to piggyback client identification to I/O requests so that server side block access history can be put into context. On the server side, we utilize the horizontal visibility graph technique to transform per-client time series of block access sequences into a connected graph for which we employ Tarjan's algorithm to disclose cut points in the connected graph. We express these patterns with feature tuples and we propose the *X*-step pattern matching algorithm to find a matching access pattern (i.e., a feature tuple) for a given block access history. Experimental results indicate that our newly proposed prefetching mechanism can ease client machines and their applications from the process of data prefetching, boosting client performance accordingly, and that it yields an attractive increase in data throughput as well.

Index Terms—Storage servers, distributed file systems, data prefetching, block access patterns, horizontal visibility graph

1 Introduction

Technological innovations in distributed systems have been unfolding at an accelerated pace, resulting in: (1) the prevalence of cloud computing, which provides a novel pathway for utility computing with unlimited resource flexibility, agility, and scalability; (2) widespread use of mobile devices equipped with limited computing facilities; (3) and the increasing speed of wireless networks [38]. The trend is, effectively, to keep client devices simple and push complexity to the cloud.

In turn, distributed file systems have been generally adopted as back-end storage systems to offer I/O services for parallel/distributed applications that need to process large amounts of data. To satisfy the ever-growing demands on I/O services, it is therefore crucial to optimize distributed file systems for better performance.

Data prefetching is a widely used optimization for traditional disk based file systems, where fetching data from the disk dominates the cost of read operations. Prefetching works particularly well for target applications that have regular access patterns, such as database servers or scientific computations [9], [10]. In a distributed setting, prefetching can mask both network latency and disk latency, and it has been successfully applied to a variety of modern distributed file systems, e.g., the Hadoop Distributed File System (HDFS) [1] and WheelFS [30].

Traditionally, data prefetching in distributed file systems is initiated by the client file system, where prefetch instructions are based on the recorded history of I/O operations. However, conventional prefetching schemes are not well suited for client machines that have limited memory and computing capacity, because sophisticated prefetching algorithms require considerable amount of log storage and computation power (as we will demonstrate below), which can cause negative effects on the application itself.

Moreover, it has been shown previously that even block level[1] access history, available only on the storage servers, can reveal sufficient information for boosting I/O performance. For instance, Li et al. showed that in a file server back-end storage system, where file blocks are indexed by their *inode* block, correlated blocks are apt to be requested relatively close to each other [33].

To offer an efficient prefetching approach for resource-limited client machines, this paper proposes a novel server-side prefetching technique for distributed file systems. In our previous work [11], [15], we proposed to map logical I/O operations on the client side to the physical I/O operations on the storage server in order to optimize data layout. Building on top of this work, this paper attempts to improve data prefetching by exploiting the combined information of client identification and the storage side access logs. Through analyzing per-client block access history, our

- *J. Liao is with the College of Computer and Information Science, Southwest University of China, Chongqing 400715, China, and the State Key Laboratory for Novel Software Technology Nanjing University, Jiangsu 210023, China. E-mail: liaojianwei@il.is.s.u-tokyo.ac.jp.*
- *F. Trahay is with Telecom SudParis, France. E-mail: rancois.trahay@telecom-sudparis.eu.*
- *B. Gerofi is with the RIKEN Advanced Institute for Computational Science, Japan. E-mail: bgerofi@riken.jp.*
- *Y. Ishikawa is with the University of Tokyo, Japan. E-mail: ishikawa@is.s.u-tokyo.ac.jp.*

Manuscript received 12 Dec. 2014; revised 19 Oct. 2015; accepted 25 Oct. 2015. Date of publication 30 Oct. 2015; date of current version 10 Aug. 2016. Recommended for acceptance by M. Steinder.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2015.2496595

1. Note that in our scheme *block* numbers refer to the unique identification of data chunks at the distributed file system level and not at the disks that are local to each storage server.

proposed prefetching scheme can predict block addresses of future read operations to guide reading block data in advance, and push it to the corresponding client. As a consequence, the client machines are not required to perform prefeching-relevant computation, so that they can achieve better performance. Specifically, we leverage the piggybacked client identifiers to separate block level access streams on the server side, for which we employ the horizontal visibility graph approach to transform the block access sequence to a connected network. Then, according to the principle of spatial locality [12], we classify block access patterns within certain ranges of block number offsets and express these patterns with feature tuples after pattern optimization. Finally, we propose the *X*-step pattern matching algorithm to find a matching access pattern (i.e., a feature tuple) for the block access history in the given prediction window. Consequently, the storage server is able to prefetch block data by resorting to the matched pattern, and then proactively forward the data to the relevant client file system for satisfying potential future requests on client nodes. In short, this paper makes the following contributions:

1) We piggyback client information on I/O requests to the storage servers so that predicting future accesses and prefetching block data can be performed entirely on the server side without any interference to the client file system or the application.
2) On the server side, we propose to utilize the horizontal visibility graph technique to transform a time series of block access events into a connected graph and classify block access patterns by employing the *Tarjan* algorithm, a graph theory technique to find cut vertices in a connected graph [35].
3) Introducing a data structure of feature tuples to represent fixed access patterns, we propose the *X*-step pattern matching algorithm to find matching access patterns to the observed block access history in order to predict future accesses quickly and accurately.

The remainder of this paper is structured as follows: the related work regarding I/O access prediction and data prefetching will be described in Section 2. The design and implementation details of this newly proposed prefetching mechanism are illustrated in Section 3. Section 4 introduces the evaluation methodology and discusses experimental results. At last, we make concluding remarks in Section 5.

2 Related Work

The I/O subsystem is becoming the bottleneck of high performance distributed systems, especially there are many communication intensive applications in them [8]. In order to achieve potential performance enhancements of storage systems, a variety of I/O history analysis-based I/O optimization mechanisms have been proposed previously [18], [19], [20], [21], [22]. However, existing approaches on the basis of I/O access analysis including the above mentioned ones have mostly focused on classifying I/O access patterns by analyzing either the history of applications' I/O requests [23], [24] or the track of block access events [20], [21], so that the analyzed results can be used for directing I/O optimization strategies, such as data prefetching. The mechanism of data prefetching focuses on how to forecast future possible read requests, and thus, the accuracy of request prediction is critical to the effectiveness and applicability of data prefetching.

There is a wide range of sophisticated approaches to fulfill data prefetching, which utilize hidden Markov models, neural networks or other predictive algorithms to forecast I/O operations by analyzing I/O access patterns of the application [23], [25], [26], [31], [34]. Simply speaking, these mechanisms employ specific algorithms to predict the application's future I/O access requests based on past I/O tracks for prefetching data. For example, J. Griffioen and R. Appleton presented a new method for reducing file system latency called *automatic prefetching*, which leverages a heuristic-based algorithm for analyzing the knowledge of past access events to predict future access requests without application intervention [31]. Another technique, called *informed prefetching* takes advantage of hints from the application to determine what data should be read in advance, assuming that file system performance can be improved by using the information provided by the application [27], [28]. However, this kind of prefetching mechanisms cannot make accurate decisions if there are no appropriate hints from the applications, and the inaccurate predictions result in negative effects on system performance.

Regarding data prefetching techniques on both local file systems and distributed file systems, Z. Li et al. proposed a data mining approach called *C-miner* to explore block correlations in storage systems on a local machine, so that the file system can make use of the discovered block correlations for guiding I/O optimization strategies, such as data prefetching or data movement [33]. We found that *C-miner* is an inspiring work, which highly influenced us in proposing a new prefetching mechanism, because the basic idea of *C-miner* motivated us to study block access patterns on storage servers. In fact, *C-miner* is able to discover the access patterns with certain frequency on the same blocks, but it cannot disclose the access patterns that target different blocks, even though these blocks might have the same correlations [33]. Similarly, S. Jiang et al. have implemented *DiskSeen*, which employs a frequent sequence-based pattern modeling technique to classify block access pattern, and both temporal and spatial correlations of block access events have been taken into account, for improving the sequentiality of disk accesses and overall prefetching performance [20], [21]. V. Padmanabhan and J. Mogul introduced a distributed prefetching scheme with distinct roles for the clients and servers in World Wide Web. Their observation was that the web servers, which are responsible to handle access requests from several clients, can make predictions on which files are most likely to be demanded in the near future [32]. I. Zhang [29] has implemented types of prefetching schemes to improve the performance of reading files and directories in *WheelFS*, which is a FUSE-based distributed file system that aims to offer flexible wide-area storage for distributed applications [30]. Recently, Y. Yin et al. have proposed the *IOSIG* tool based on their previous work [23], which can keep track of parallel I/O calls of an application and then analyze the collected information to provide a clear understanding of I/O behavior of the application on the client machine [17]. As a consequence, the client file systems can issue prefetching requests or adjusting layout requests to the storage servers

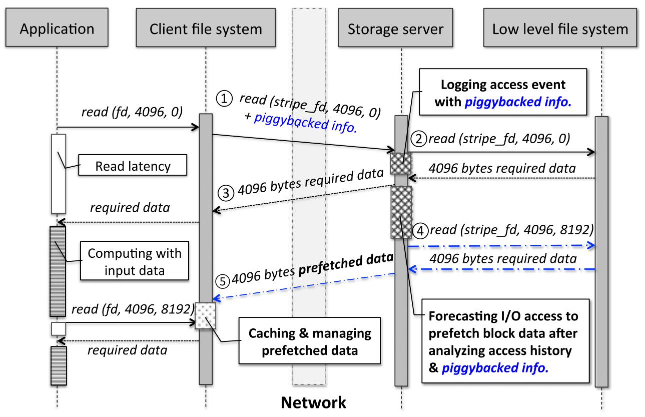

Application
Client file system
Storage server
Low level file system
read (fd, 4096, 0)
① read (stripe_fd, 4096, 0) + piggybacked info.
Logging access event with piggybacked info.
② read (stripe_fd, 4096, 0)
4096 bytes required data
Read latency
③ 4096 bytes required data
required data
④ read (stripe_fd, 4096, 8192)
4096 bytes required data
Computing with input data
⑤ 4096 bytes prefetched data
read (fd, 4096, 8192)
Caching & managing prefetched data
Forecasting I/O access to prefetch block data after analyzing access history & piggybacked info.
required data
Network

Fig. 1. Overview of the proposed server-side data prefetching mechanism. The assumed synopsis of a read operation is *read(int fildes, size_t size, off_t off).*

for I/O optimization after certain access predictions. However, tracing and analyzing I/O calls on the client node causes extra space and time overhead, thus the *IOSIG* tool may not be a good choice for configuration-limited client machines to conduct I/O optimization operations.

Furthermore, C. Amza's group [36] and X. Zhang's group [37] are the pioneers of storage server side prefetching in network based file systems. Both groups proposed their prefetching schemes running on storage servers, and their evaluation verified the effectiveness of server-side prefetching. These schemes, however, either require modifications of the applications or are only working for a very limited number of block access patterns. In brief, although block access history reveals the behavior of disk traces, there are no general storage server-side prefetching schemes that analyze block access history in a distributed file system for yielding better system performance. The well-known reasons for this are the difficulties in modeling block access history to generate block access patterns, as well as the aporias in deciding the destination client file system for pushing the prefetched data from storage servers.

3 Data Prefetching on Storage Servers

In general, the sequence of block access on the storage server is ordered in time, so that the block access sequence can be split into successive parts by a constant time interval, meaning that the sequence resembles typical time series [13], [34]. This is a crucial fact for understanding the proposal of this paper, which is a server-side data prefetching mechanism that considers block access history on the storage servers as a time series, and then tries to classify various block access patterns from the series. Consequently, it predicts the future block access requests by matching the fixed access patterns with current block access events in the prediction window to guide reading block data in advance, and finally the fetched data will be pushed to the relevant client file systems to fulfill potential I/O requests on the client side.

Fig. 1 presents the basic idea of our proposed server-side prefetching mechanism, where the interaction between the client file system and storage server can be described through the following steps:

1) After contacting the metadata server to know which storage server needs to be accessed for the actual data request, the client file system sends the corresponding storage server an I/O request accompanying with some additional information about the application and the client file system (labeled as *piggybacked info.*).
2) The storage server, fetches the requested data from the low level file system, and records the block access event along.
3) The application on the client machine can perform computing tasks after receiving the required data from the storage server.
4) Meanwhile, the storage server is able to forecast future block I/O access requests by analyzing the history of block I/O access events and the piggybacked client information. Therefore, the storage server can issue relevant physical read requests to the low level file system for reading data (that are predicted to be accessed by the future I/O requests) in advance.
5) Finally, the prefetched data are forwarded to the corresponding client file system (determined by the piggybacked client identification) proactively. As a result, when the prediction of I/O access is successful, the buffered data on the client file system can be returned to the application instantly, and then read latency can be reduced to a great extent.

In brief, the storage server can predict the future block I/O access by referring the matched pattern, and then directly issue a low-level read request to fetch the data in advance after analyzing the piggybacked information sent by the client file systems. The fetched data will be eventually pushed to the associated client file system for satisfying potential application I/O requests. The specifications of the piggybacking mechanism, which is leveraged to map client I/O requests to the block access events on the servers, are presented in Section 3.1; Section 3.2 describes the prefetching algorithm; the implementation details are explained in Section 3.3.

3.1 Piggybacking Mechanism

In our proposed server-side prefetching mechanism, the knowledge of mapping logical I/O requests on client machines to the actual physical block access on storage servers is an obligatory precondition. In other words, the storage servers need to know certain information about client file systems and applications. Although we have proposed a mapping mechanism in our previous work, it requires the client file systems to keep track of logical access information and send the logs to the server side [15]. To reduce the overhead resulted by client logging, in this paper we leverage a piggybacking mechanism, illustrated in Fig. 2, to transfer the related information from client file systems to storage servers for contributing to build the mapping relationship between logical access and physical access. As it is described in Fig. 2, when sending a logical I/O request to the storage server, the client file system piggybacks the information about the client file system and the application with the request. As a consequence, the history of block access events can be put into a client specific context so that the storage servers can perform prediction of future block

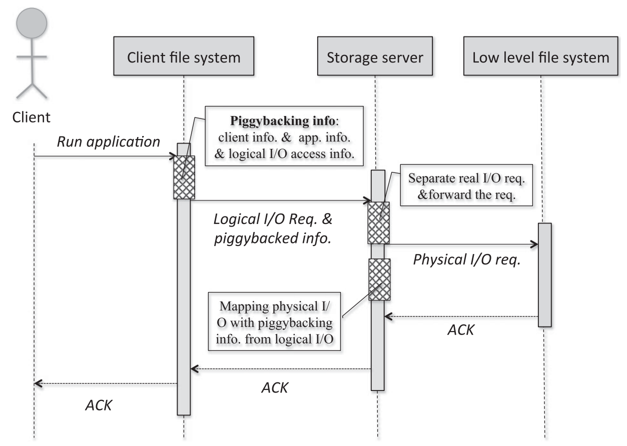

Client
Client file system
Storage server
Low level file system
Run application
Piggybacking info: client info. & app. info. & logical I/O access info.
Separate real I/O req. &forward the req.
Logical I/O Req. & piggybacked info.
Physical I/O req.
Mapping physical I/O with piggybacking info. from logical I/O
ACK
ACK
ACK

Fig. 2. The piggybacking mechanism for mapping client I/O requests to block access events occurred on the servers.

access requests, and then push the prefetched data to the relevant client file systems proactively.

To describe this mechanism from another angle, the client file system is responsible for keeping extra information about the application and client file system; it then piggybacks the extra information to client I/O requests, and sends them to the corresponding storage server. On the other hand, the storage server parses the request to separate the piggybacked information and the real I/O request from the client request. Apart from forwarding the I/O request to the low level file system, the storage server has to log the disk I/O access event with the piggybacked information regarding the corresponding logical I/O access. As a result, the storage server is able to make a record for each block access event accompanying the client context piggybacked by the corresponding client I/O request. Fig. 3 details the actual data fields that are included in each log record in our proposed prefetching scheme. As the figure shows, the field of *client file system info* in the piggybacked information contains the IP address, and the ID of client file system, etc., which are required for pushing prefetched data to the corresponding client file system.

3.2 Prefetching Algorithm

Our prefetching mechanism intends to identify block access patterns on the storage servers for guiding read operations in advance. In this section, we first demonstrate the effectiveness of block access patterns, and then discuss the details of the prefetching algorithm.

3.2.1 Effectiveness of Block Access Patterns

Client I/O requests that belong to the same application usually have certain logical relationship, and there are many

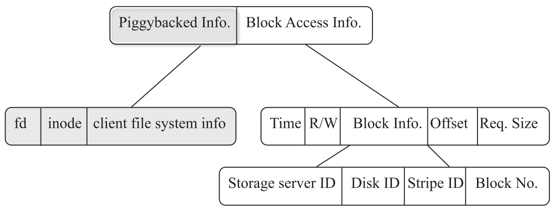


Fig. 3. Details of the logged information attached to server side block accesses.

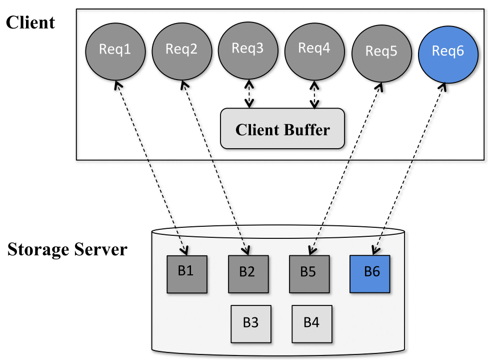


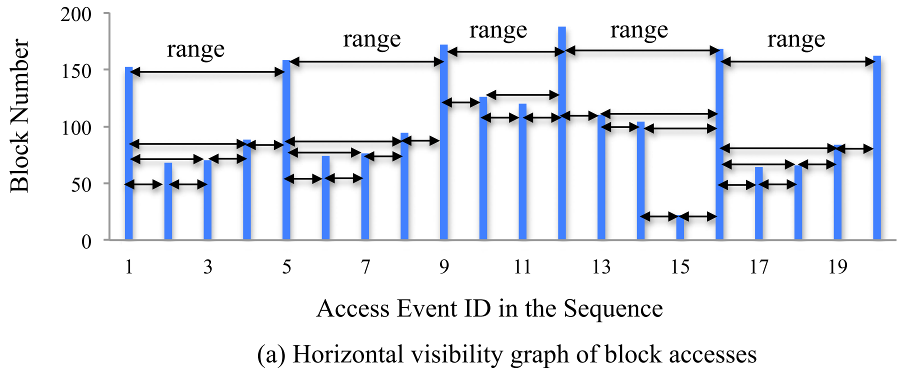
Fig. 4. Relationship between logical and block level access patterns.

client side prefetching techniques which exploit this observation, e.g., *IOsig+* [17], [23]. On the other hand, block level access patterns usually lack the logical connection to application context, which makes it difficult to use them for directing data prefetching in practice. We have conducted multiple case studies and experiments, and found that block access patterns may have certain relationship with logical access patterns occurred on the client side, though they are not a one-to-one relationship.

By referring to Fig. 4, one can clearly see why block access patterns are effective to guide data prefetching. Assume that there is a client access pattern *[Req1, Req2, Req3, Req4, Req5, Req6]*, in which the requested data of *Req3* and *Req4* are satisfied with the data buffered in the client cache. We can easily predict the future access request, i.e., *Req6* for directing client side data prefetching when *Req1*, *Req2*, *Req3*, *Req4*, and *Req5* have been observed. On the other hand, assume that we can classify *[B1, B2, B3, B4, B5, B6]* as a block access pattern on the storage server for the aforementioned client access pattern when it appears first (i.e., there is no cached data at that time). In case two of the client requests have been already satisfied with the cached data, there corresponding block access pattern on the storage servers will be *[B1, B2, B5, B6]*. It is then possible to forecast that there might be another block access request to *B6* in the future in case the observed access sequence was *B1*, *B2*, and *B5*; thus, it is possible to guide prefetching the data of block *B6* in advance from the viewpoint of the storage server.

3.2.2 Pre-Processing Time Series of Block Access

For the purpose of providing basic data about block access events to classify access patterns, we use the horizontal visibility graph technique, which was proposed to generate mapping networks from time series [14]. A sequence of observed block access events can be transformed to a connected graph, in which a node is utilized to represent a block access event, for modeling block access patterns. Specifically, a data point in the time series of block access events can be expressed as a node of $(time, offset, size)$, which indicates the access request arrived at *time*, starting at address of *offset* and with a request size of *size*. Because *offset* is the most important field in an access event, we set the value of each node as the *offset* of the corresponding access event. Two nodes in the horizontal visibility graph are connected when one can draw a horizontal line in the time series. Namely, all nodes must follow the connection rule, which is defined in *Definition* 1, to decide whether a node can be



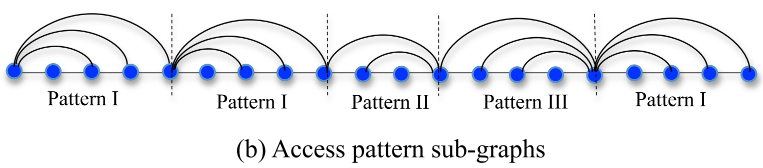


Fig. 5. Horizontal visibility graph of block access events and their corresponding access patterns.

connected with another one or not. Fig. 5a illustrates an example of a horizontal visibility graph that is transformed from a given time series of block access events. Fig. 5b shows how to present the horizontal visibility graph by employing a connected graph, because we intend to use advanced approaches or algorithms in graph theory to classify access patterns and perform pattern matching subsequently.

Definition 1. *Connection Rule: when* $node(t_a, off_a, size_a)$ *is connected with* $node(t_b, off_b, size_b)$ *in the visibility graph, if and only if for an arbitrary* $node(t_c, off_c, size_c)$ *that occurred between the aforementioned two nodes having* $off_a > off_c$ *and* $off_b > off_c$ *when* $(t_a < t_c < t_b)$.

3.2.3 Classifying and Modeling Access Patterns

After transforming a sequence of block accesses into a horizontal visibility graph, we can classify access patterns and represent them as sub-figures. Therefore, extracting block access patterns from the history of block access events can be fulfilled by identifying sub-graphs in the corresponding horizontal visibility graph. Besides demonstrating how block access patterns can be helpful for data prefetching, this section also discusses the approach to classify access patterns introducing a novel data structure for access pattern management.

(1) Classifying Access Patterns. According to the locality of reference, which suggests that access events which belong to the same client request are likely to occur within a limited block range, we propose a range-based approach to identify access patterns in the horizontal visibility graph. Specifically, the process of exploring access patterns can be addressed by the following two steps:

Step 1: Identifying Access Patterns. To identify access patterns from the horizontal visibility graph including all observed block access events, we utilize a modified version of the *Tarjan* algorithm [35], which pinpoints cut vertices in a connected graph, that corresponds to the horizontal visibility graph. The basic idea of the *Tarjan* algorithm is to employ *DFS* (Depth First Search). First, we follow vertices in a tree representation called the DFS tree. In the *DFS* tree, a vertex u is parent of another vertex v, when v is disclosed by u (obviously v is adjacent of u in graph). Finally, in the *DFS* tree, a vertex u is a cut point when one of the following two conditions is satisfied:

1) u is the root of *DFS* tree and it has at least two children.
2) u is not the root of *DFS* tree, but it has a child v, and this node has a back edge to one of the ancestors (in *DFS* tree) of u, moreover, no vertex in the sub-tree is rooted with v.

As a consequence, we can obtain a number of sub-graphs divided by the cut vertices with the time complexity of $O(V + E)$, where V is the number of nodes, and E is the number of edges. We define as the *identified access pattern*. a connected sub-graph with several nodes in the horizontal visibility graph

Step 2: Optimizing Access Patterns. In the first step described above, it is possible to obtain many independent or repeated access patterns. For the purpose of refining them, the second step intends to optimize and extend those patterns. To this end, we check the access pattern that follows each occurrence of an identified pattern, and attempt to extend it. As illustrated in Fig. 6, when *pattern I*'s next pattern is always *pattern II*, the later one can be integrated into the former one. Thus, a new access pattern, i.e., *pattern IV* is created to replace two adjacent patterns.

Furthermore, to accelerate the speed of access pattern matching and to improve prediction accuracy when conducting server-side prefetching, it is better to set a range for the number of block access events in a fixed access pattern. That means the patterns should be shifted if the numbers of

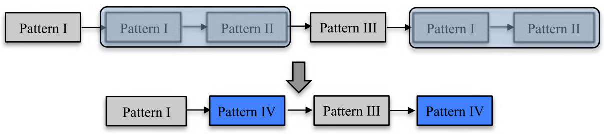


Fig. 6. Extending existing access patterns.

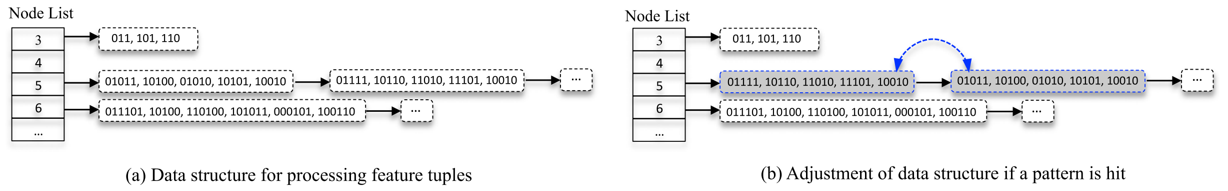


Fig. 8. Data structure for managing feature tuples and its relevant adjustment.

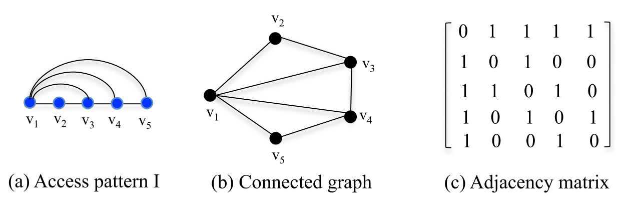
access events in the patterns are out of the pre-defined range. Besides, no extension to the access patterns should be performed if the number of access events in the extended pattern would exceed the upper limit.

(2) Representing Patterns as Feature tuples. In order to model access patterns for quick and accurate pattern matching, we have introduced a novel data structure, derived from a matrix-based mechanism, and named it as *feature tuple*. The data structure definition of a feature tuple can be found in *Definition* 2.

Definition 2. *A feature tuple of an adjacency matrix of the sub-graph is defined as a vector with the elements:* [Num_{node}, $\mathrm{Sum}_1, \mathrm{Sum}_2, \ldots, \mathrm{Sum}_N$], *where Num_{node} indicates the number of total nodes in the connected sub-graph, i.e., the number of access events in the pattern, and Sum_i is a binary value related to the connection relationship among the elements in* Row i *of the adjacent matrix.*

As a matter of fact, the data structure of feature tuple can reflect the connection structure of an identified sub-graph, which is derived from its corresponding access pattern. Moreover, a feature tuple can be used to complete pattern matching with less time complexity and it requires less space for storing the adjacency matrix of a connected graph. Fig. 7 demonstrates the mechanism of mapping an access pattern to the corresponding adjacency matrix by employing a connected graph to represent the fixed *Access pattern I*. For example, according to the Definition 2, *Access pattern I* can be represented by the feature tuple of *[5, 01111, 10100, 11010, 10101, 10010]*.

After analyzing the occurred block access events, we can obtain multiple valid access patterns and their relevant feature tuples. For the purpose of matching access patterns effectively, feature tuples are managed as a linked list, which can be quickly adjusted on the basis of the matching frequency of access patterns. Fig. 8a demonstrates the data structure for storing the feature tuples of fixed access patterns. The element in the *Node List* array indicates the number of nodes in the access pattern, and the linked list belonging to an element in the array means the access patterns that involve the same number of I/O events. Moreover, for boosting matching accuracy and reducing the matching overhead, the linked list can be adjusted dynamically. For instance, an adjustment case of the data structure is illustrated in Fig. 8b. When the access pattern of *[5, 01111, 10110, 11010, 11101, 10010]* has been hit recently, the pattern should exchange its position with the previous one. Owing to this dynamic adjustment mechanism, the most frequently hit patterns are located in front of the list, which accelerates pattern matching.



$$\begin{bmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Fig. 7. Mapping an access pattern to its corresponding adjacency matrix.

3.2.4 The X-Step Matching Algorithm

As we discussed before, data prefetching is a widely used technique for hiding data access latency by referring to identified access patterns. However, the effectiveness of prefetching is primarily dependent on the prediction accuracy of future access requests [16], [33]. Furthermore, the speed of predictions on future requests is also critical to the effectiveness of the prefetching mechanism. Therefore, in order to obtain attractive performance improvements brought by server-side data prefetching, we have introduced a novel pattern matching algorithm, which we call the X-step matching algorithm. X-step compares the feature tuple of an observed block access with the feature tuples of previously identified access patterns.

Algorithm 1 illustrates the process of the X-step matching mechanism in detail, which can forecast X successive block access events that might occur in the near future. The basic idea of this algorithm is to compare the block correlations among access events in the given prediction window, in which there are a number of access events utilized for conducting prediction (labeled as *cur_tuple*), with the block correlations among the first few access events of the identified access patterns (labeled as *tuple*). By referring to Fig. 8a, there is an example that shows how to forecast two subsequent access events by using the presented X-step prediction algorithm, when there are three access events in the prediction window. First, it is necessary to build the correlations among the three access events in prediction window, and we assume that these three events are strongly connected in the graph, i.e., *cur_tuple* is *[3, 011, 101, 110]*. Then, we need to locate the list of fixed patterns, in which each pattern has *5* (i.e., *3+2*) access events in total. Next, we conduct a right shift of two bit positions on each element of the tuple corresponding to the identified pattern in the list and compare the elements in *tuple* with the elements in *cur_tuple*. Obviously, the first tuple in the list with five nodes is not a match to *cur_tuple*, so we proceed with checking the subsequent ones in the same way. Finally, the algorithm will finish only if the match is found or the entire list has been traversed.

We have also studied the time complexity of the newly proposed matching algorithm: since it is supposed to traverse the linked list, in which all access patterns have the expected number of involved access events. Consider that we have a total of M identified access patterns, and the largest number of access events in the pattern is N. The newly proposed algorithm will require $O(MN)$ time overhead, and $O(MN)$ extra-space in the worst case. In fact, most of space is used to store the feature tuples of the already identified access patterns.

Algorithm 1. X-Step Matching Algorithm

```
Initialization:
    1) available node_list with the feature tuples of the fixed
       access patterns;
    2) the built cur_tuple of the current block access events in
       the prediction window;
    3) int num = 0; int i= 0;
Iteration:
 1: while $true$ do
 2:   $num = get\_number\_of\_node(cur\_tuple)$;
 3:   /*try to match the patterns having num + X access */
 4:   $num = num + X$;
 5:   if $num$ is not in the range of $node\_list$ then
 6:     return null; /* no matched pattern */
 7:   end if
 8:   while $node\_list[num] \rightarrow next \neq null$ do
 9:     $tuple = node\_list[num] \rightarrow next$
10:     /*comparing first $num$ events of $tuple$ with all events
        of $cur\_tuple$*/
11:     while $((tuple[i] >> X)) == cur\_tuple[i]$ do
12:       if $++i == num$ then
13:         return $tuple$; /*matched pattern found*/
14:       end if
15:     end while
16:     $tuple = tuple \rightarrow next$; /*try next pattern*/
17:     $i = 0$;
18:   end while
19:   return $null$; /*no matched access pattern*/
20: end while
Output:
    null or the matched access pattern, i.e., tuple.
```

3.3 Implementation

We have implemented the proposed prefetching mechanism in the PARTE file system [11]. PARTE has been implemented from scratch and it provides POSIX support. The implementation has three modules running at the user level:

- *partemds*: the module of active metadata server, which aims to offer metadata service, and it does not have any relationship with data prefetching.
- *parteost*: the module of storage server is responsible for the management of real file data; moreover, the server-side prefetching are also implemented in this module.
- *partecfs*: the module of client file system has been designed and implemented based on FUSE [6]. The client file system buffers and manages prefetched data, and returns them to applications' I/O requests.

As a matter of fact, our implementation of server-side prefetching scheme in PARTE aims to illustrate the feasibility and applicability of the proposed mechanism. For fairness of comparison in experiments, we have also implemented other prefetching schemes in the PARTE file system.

TABLE 1
Node Specification of the Storage Servers and the Clients

	Storage Servers	Clients
CPU	2xIntel E5410 2.33GHz	Intel E5800 3.20GHz
Memory	1x4GB 1066MHz/DDR3	4GB DDR3-SDRAM
Disk	6x114GB 7200rpm SATA	500GB 7200rpm SATA
Network	Intel 82598EB, 10GbE	1000Mb or 100 Mb
OS	Ubuntu 13.10	Debian 6.0.4

4 Evaluation

This section describes the experimental methodology for evaluating the proposed prefetching mechanism, and reports the experimental results. First, we describe the experimental setup; next, the experimental results about both positive and negative effects brought about by different prefetching schemes are presented to show the applicability of server-side prefetching. Then, we analyze the prediction accuracy caused by the server-side prefetching mechanism and report the relevant results. At last, we conduct a case study with real block traces to illustrate the feasibility of the newly proposed prefetching mechanism in practice.

4.1 Experimental Setup

4.1.1 Experimental Platform

We employed one cluster and two local area networks (LANs) to conduct evaluation experiments. One active metadata server with four storage servers of the distributed file system are deployed on a five nodes server cluster, and all client file systems are installed on 12 nodes of two separate LANs. Specifically, for the purpose of emulating a distributed computing environment, six of the client file systems were installed on a LAN that is connected with the server cluster by a 1 GigE Ethernet; while the other six client file systems were installed on a LAN that is connected with the cluster by a 100 M Ethernet. Table 1 shows the specifications of nodes on the server cluster as well as on the two LANs. All clients are equipped with MPICH2-1.4.1.

4.1.2 Prefetching Configurations

The following prefetching configurations have been used for evaluation.

- *Non-prefetching* indicates a distributed file system without any data prefetching mechanisms, i.e., the original implementation of PARTE. In other words, there is no block access tracing and no analyzing facility is enabled in the file system.
- *Readahead* prefetching is a typical sequential prefetching mechanism on the storage servers. If there are non-consecutive misses to the server, the file system is supposed to issue a prefetch request to read certain consecutive blocks in advance [33]. We fixed the number of prefetched blocks in our evaluation experiments as 4 and 32, respectively.
- *Signature-based* prefetching is a typical data prefetching scheme on the client file systems that analyzes the occurred I/O requests of applications. Our implementation in PARTE is based on the source code that is publicly available [2]. This prefetching approach

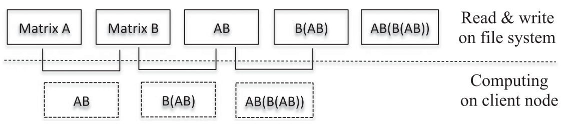


Fig. 9. The algorithm of matrix multiplication in our benchmark: there are three consecutive multiplications, and the intermediate results are flushed back to the file system. In other words, all input data should be read from the file system.

allows users or client file systems to characterize access patterns of an application in two steps: 1) a trace collecting tool gets the trace of all the I/O events of the application; 2) through offline analysis of the traces, the analyzing tool determines the I/O signature to guide data prefetching on the client side [17], [23].

- *Server-side* prefetching is our newly proposed mechanism. It performs block access prediction to guide data prefetching on storage servers. This mechanism employs several sophisticated algorithms in graph theory to manage access patterns and to accomplish pattern matching.

4.1.3 Benchmarks

To show the merits resulted by our proposed server-side prefetching technique, as well as the overhead caused by tracing and analyzing block access history to predict future block access requests, the following benchmarks were employed in the evaluation experiments:

- *Sysbench* is a commonly used multi-threaded benchmark tool for evaluating OS parameters that are critical for a system running a database under intensive workloads [3]. *Sysbench* contains certain programs and each program aims to explore the performance of the specific aspect under Online Transaction Processing (OLTP) workloads. Thus, we utilized this benchmark to measure transaction throughput and I/O response time used for handling online transactions.
- *IOzone* is a popular micro-benchmark, and widely used to evaluate the performance of a file system by employing a collection of I/O access patterns, such as sequential, random, reverse order, and strided [4]. Therefore, we used it to measure read data throughput of the file system with different prefetching schemes when there are various type of access patterns.
- *Matrix multiplication* is a test computation program developed by ourselves. It reads the matrix elements from files stored on the distributed file system to perform multiplication tasks, and Fig. 9 demonstrates its work flow clearly. After running the benchmark for multiplying the matrices with varying sizes, we measured the time needed for completing the multiplication tasks to show whether the newly presented prefetching scheme can save computing power on the client nodes or not, as well as how much benefits can be yielded by the proposed prefetching scheme if it can save certain resource usage.

4.1.4 Parameter Settings

In all experiments, considering that the client nodes might be resource-limited, size of the prefetching buffer on the client was configured to *128* blocks (indicating that the client file system can cache *128* blocks of data), for which the Least Recently Used (LRU) replacement policy was used to replace cache blocks. The range of block access events in the fixed access patterns was set as *[8, 16]*. The value of X in the X-step matching algorithm was set as *4*, which means we tried to forecast *4* successive block access events, and then read 4 block data in advance. As a consequence, the size of the prediction window is calibrated to stay in the range of *[4, 12]*.

After running multiple experiments, the size of slides window in the sequence of block access, which is used for classifying access patterns, was configured to *8* times the lower boundary of the range of block access events in the patterns. In other words, we set the range of block access events in the fixed access patterns as *[8, 16]*, which indicates the size of slides window is consequently fixed as $8 \times 8 = 64$. Therefore, the newly proposed prefetching scheme does not start modeling access patterns until there are *64* occurred block access events at the beginning of the execution for directing data prefetching. Meantime, during the execution of the application, our scheme will model block access patterns again (i.e., refreshing block access patterns) if the recently occurred *64* block access events do not match any fixed access patterns. Besides, the maximum number of block access events used for modeling patterns is fixed as *2,048*, which means only the recently occurred *2,048* block access events are taken into account for classifying access patterns.

4.2 Experimental Results

This section reveals both positive and negative effects induced by the newly proposed server-side prefetching mechanism.

4.2.1 I/O Response Time

First, we tested Sysbench-0.5 multi-table OLTP with MySQL-5.6.10 [5] as the back-end database. In the experiments, each client machine had its own database, and the configuration was eight tables with total 10 G of data and around 20 million tuples; besides, the I/O capacity was set as 1,000. Fig. 10 presents the experimental results about transaction throughput and I/O response time specifically when the running modes are read only and read/write. In the legend of the figure, *Readahead-4* and *Readahead-32* means prefetching 4 and 32 blocks of data, respectively, when employing the *Readahead* prefetching.

The results reported in the figure show that prefetching mechanisms are beneficial for database-relevant applications, because we could achieve better system throughput, i.e., transactions per second, as well as lower I/O response time. Moreover, in contrast to *Readahead* prefetching and *Signature-based* prefetching, the newly proposed *Server-side* prefetching was able to yield the lowest I/O response time and the highest transaction throughput in the majority of cases, in which there were a varying numbers of threads on every client machines, and all threads issued I/O requests to the file system in parallel.

4.2.2 Data Throughput

After evaluating the effectiveness of the proposed prefetching mechanism when executing database-related applications, this section aims to measure the read data throughput by

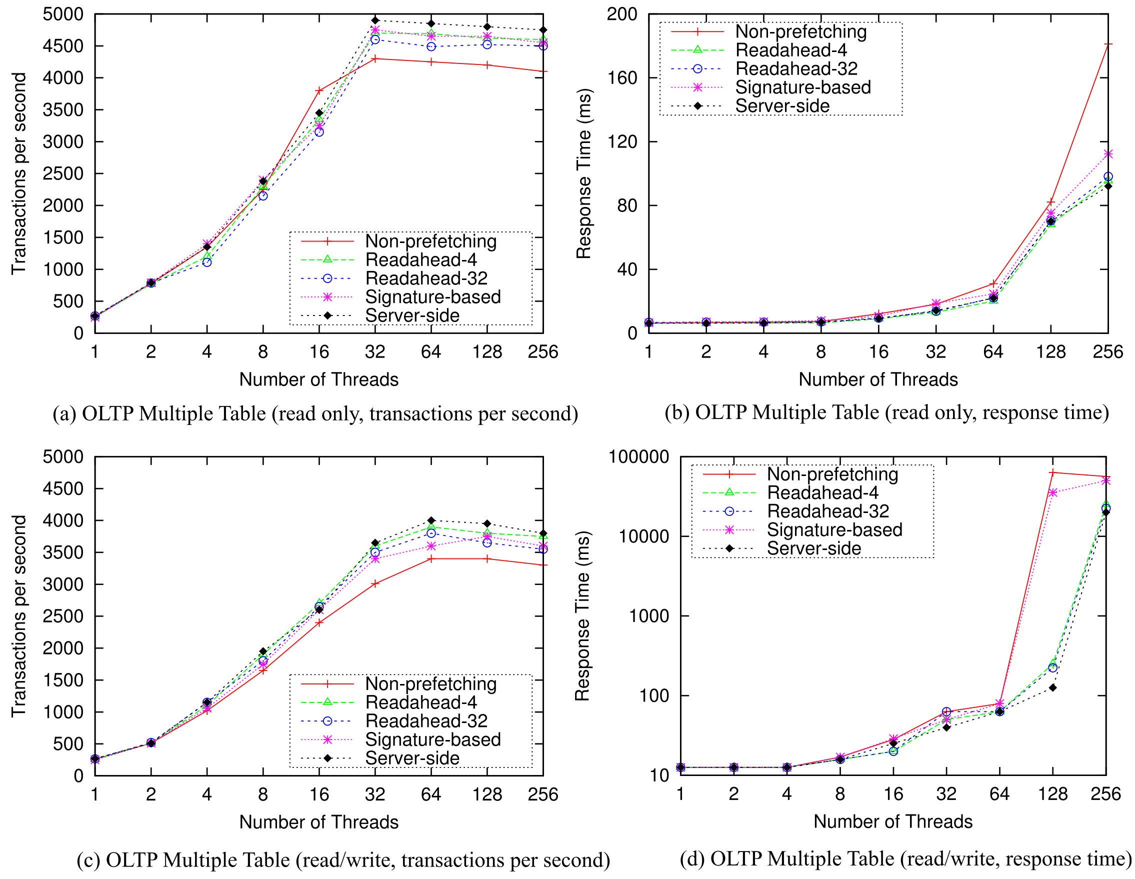


Fig. 10. Throughput and I/O response time results when running *Sysbench*.

running a parallel I/O benchmark, i.e.,*IOzone*. Although *IOzone* has several read and write patterns to evaluate the performance of file systems from specific aspects, we only focus on access patterns that are suitable for read operations because prefetching only benefits read requests.

Fig. 11 shows the read data throughput reported when running the *IOzone* benchmark with various access patterns. The experimental results show that the proposed prefetching scheme performed the best in most cases when running the *IOzone* benchmark. Especially, when conducting both *backward reads* and *stride reads*, for which the corresponding results of read data throughput are demonstrated in Figs. 11b and 11c, respectively, the *Server-side* prefetching scheme achieved from 16.29 to 162.37 percent higher read data throughput, in contrast to the *Readahead* and the *Non-prefetching* schemes. Moreover, the *Signature-based* prefetching scheme, which is a typical client-side prefetching scheme, did not work better than the *Non-prefetching* scheme in most of the cases, which we believe is due to the fact that it does not have any means to classify access patterns on client I/O requests or any practical implementation of this notion [17].

4.2.3 Accelerating Computation

We emphasized that server-side prefetching can alleviate client machines from the process of data prefetching. In order to illustrate the benefits for client machines resulted by server-side prefetching, we executed a matrix multiplication benchmark, and recorded its execution time. Fig. 12 reports the time required for performing the matrix multiplication, where the input data was stored on the distributed file systems running with different prefetching schemes. The Figure shows detailed breakdown of the execution time, including the time needed for performing computation, the time required for I/O operations, and the time used for predicting future I/O requests on the client side, if applicable. Since only the *Signature-based* mechanism performs client-side request prediction, it is the only mechanism for which the forecasting time on the client side is non-zero.

The results show that the newly presented server-side prefetching scheme required the shortest time for conducting multiplication tasks when the matrix size is greater than 200. For instance, if the matrix size is 1,000, which means there are 1,000 $\times$ 1,000 matrix elements, the *Server-side* prefetching scheme could reduce the completion time by more than 24.75 percent compared to *Non-prefetching*, *Readahead-4* and *Signature-based* prefetching. The results show that while the computation time remains constant for all the prefetching mechanisms, the time spent on performing I/O varies greatly. Consequently, the *Server-side* prefetching mechanism, which achieves the greatest I/O time reduction, improves the overall performance of the application.

Another interesting observation revealed in the figure is that *Readahead-32* prefetching worked better than *Readahead-4* prefetching. Because the matrix multiplication benchmark reads input data regularly, the more data are cached on the client machine the greater benefit becomes. In other words, *Readahead-32* fetches data from the server eight times less often than *Readahead-4*. The I/O latency for *Readahead-32* is

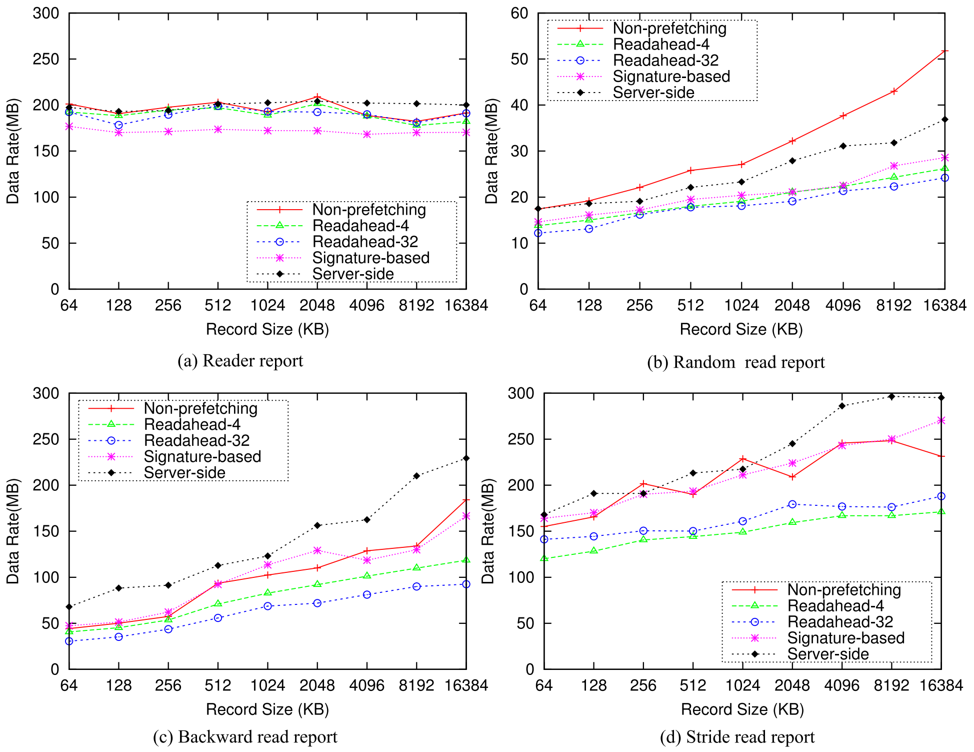


Fig. 11. Read data throughput on various access patterns measured by IOzone.

therefore greatly reduced. Furthermore, for the *Signature-based* prefetching scheme, the overhead caused by the prediction mechanism on the client-side is such that there is only little gain compared to the *Non-prefetching* policy. The matrix multiplication application reads the matrix elements regularly, so that the percentage of prediction hits is high, but the benefits brought by client-side prefetching could merely compensate the overhead of conducting prediction on the client machines. As a result, *Signature-based* prefetching performed worse than the *Server-side* prefetching mechanism that we propose.

4.2.4 Server Side Overhead

We have demonstrated that server-side prefetching can effectively and practically forecast future disk read operations to guide data prefetching for different workloads, but it is also necessary to measure the overhead resulted by the proposed prefetching scheme. Table 2 illustrates the execution time and space overhead on the storage servers caused by the newly presented prefetching technique. Because *IOzone* is a representative I/O benchmark to test I/O performance of a storage system, prefetching and transferring the prefetched data cause more than 9.3 percent of time overhead. However, the overhead due to our proposed mechanism for compute-intensive applications and OLTP workloads remains limited so that it does not degrade the performance on the server-side. For example, when the workload is matrix multiplication, the time required for predicting future disk operations to guide data prefetching and then forward the data to the corresponding client file systems is around 3.0 percent of the total processing time on the storage servers. Namely, a major part of processing time can be devoted to handle I/O processing; therefore, the server-side prefetching mechanism is still practical for storage systems in distributed computing environments, though it indeed introduces some overhead.

Furthermore, the consumed space for storing disk traces to conduct I/O prediction is reported in Table 2. The newly presented server-side prefetching is efficient in terms of space overhead for the majority of traces. Though running the *IOzone* benchmark resulted in more than 100 MB but less than 170 MB tracing logs, as it is a typical I/O

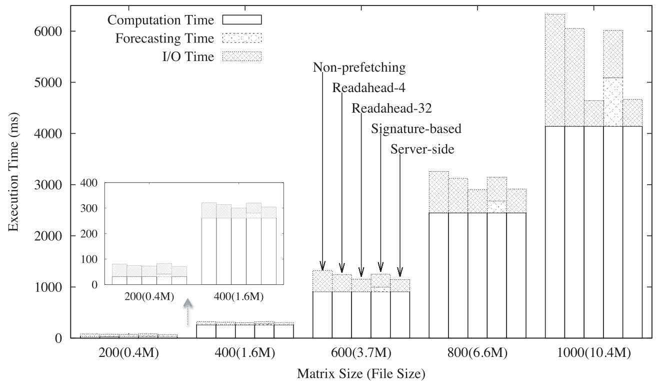


Fig. 12. Time required for executing matrix multiplications.

TABLE 2
Server-Side Prefetching Overhead

Benchmark	Time (%)	Space (MB)
IOzone	9.18	168.32
Sysbench	3.22	81.08
Matrix (400×400)	3.26	0.66
Matrix $(1{,}000 \times 1{,}000)$	2.17	4.22

TABLE 3
Prediction Hit Ratio

Benchmark	Reads	Predictions	Hits (Percentage)
IOzone:read	2,097,184	116,158	940,766 (80.99%)
IOzone:backward-read	1,048,576	961,672	840,329 (87.38%)
IOzone:stride-read	1,048,576	961,680	840,332 (87.38%)
IOzone:random-read	2,097,152	219,221	105,423 (48.09%)
Sysbench:read& write	3,326,112	2,098,463	1,729,863 (82.43%)
Sysbench:read-only	2,372,024	1,319,833	1,073,502 (81.33%)
Matrix (200×200)	91	27	26 (100 %)
Matrix (400×400)	355	291	289 (96.29 %)
Matrix (600×600)	795	784	781 (99.61 %)
Matrix (800×800)	1,410	1,346	1,334 (99.11 %)
Matrix (1000×1000)	2,201	2,137	2,127 (99.53 %)

benchmark and focuses on I/O operations rather than computing tasks. The results show that less than 82 MB space was used for storing disk traces to forecast future disk I/Os when the benchmark was *Sysbench* (i.e., the OLTP workloads). In a word, analyzing disk traces and prefetching block data can run on the same machine as the storage system without causing too much memory and disk overhead. For long-running applications, the size of trace logs may become extraordinary large, in this case, *Server-side* prefetching may discard certain logs that occurred earlier, because disk block correlations are relatively stable during a given period, and disk access logs from earlier stages are not instructive for forecasting future disk access events [33].

4.3 Prediction Analysis

After reporting the merits and demerits brought by the server-side prefetching scheme, we present the prediction error/deviation when applying our estimation models on the different benchmarks, to show the feasibility of the prediction algorithms. Therefore, we also counted the number of prediction hits, the numbers of predictions and total occurred read operations, when running the benchmarks.

The experimental results of related statistics are elaborated in Table 3. From the reported data in the table, we can conclude that our prediction model is not capable of predicting all future read operations on the disk. Namely, only when the current block I/O events in the given prediction window can match part of the block access events in a fixed access pattern, will the read predictions be performed. The proposed prediction algorithm can achieve acceptable prediction hits, though as one would expect, it attains worse prediction of future block accesses if the access pattern is random. In summary, the proposed prediction algorithm is able to yield preferable prediction accuracy, so that it can result in better system performance, which had been also proven in our evaluation experiments.

4.4 Case Study with Real Block Traces

For conducting a more objective evaluation to verify the applicability and feasibility of the proposed mechanism in practice, we selected several real block traces published by Storage Performance Council (SPC) consisting of *Financial* traces and *WebSearch* traces [7]. The *Financial* traces are from OLTP applications at two large financial institutions, which are relatively sequential, and *WebSearch* traces are generated by a popular search engine, which are relatively more random. Table 4 presents a summary statistics of these traces about read events, as the prefetching mechanism only helps with read requests. To be specific, the total number of requests, the number of read events, the percentage of read events, and the average read size in blocks of each block trace are reported.

TABLE 4
Workload Characteristics of the Selected Traces

Trace	# of Requests	Reads (Percentage)	Average read size (blocks)
Financial1	5,334,987	1,235,633 (23.16%)	4.50
Financial2	3,699,194	3,046,112 (82.35%)	4.56
WebSearch1	1,055,448	1,055,236 (99.98%)	30.29
WebSearch2	4,579,809	4,578,819 (99.98%)	30.14
WebSearch3	4,261,709	4,260,499 (99.97%)	30.81

In the experiments, the client file system issues I/O requests to the storage servers for writing/reading block data, according to the records in the traces. Fig. 13 shows the average reponse time to read requests in the selected five block traces. The write requests are not considered here since the prefetching schemes do not benefit write requests. In the figure, the X-axis shows the name of the selected block trace, and the Y-axis illustrates the average I/O response time (the lower the better). As seen, our proposed prefetching mechanism outperforms others because of its accurate block access prediction, which shortens response time of the application I/O requests. To put it from another angle, *Server-side* prefetching decreases the average I/O response time by 24.18-34.07 percent in contrast to the commonly used *Readahead* prefetching, and 18.92-20.22 percent compared to *Signature-based* prefetching. Furthermore, the experimental results also illustrate that *Readahead* prefetching did worse than *Non-prefetching* for the *WebSearch* traces. That is because less than 3 percent of the access events from the *WebSearch* traces are sequential. As a result, the consecutively fetched data are rarely used for responding to applications requests, which places negative effects on the system performance.

5 Concluding Remarks

This paper has proposed, implemented and evaluated a server-side prefetching mechanism, which does not require any client file system or application side participation in

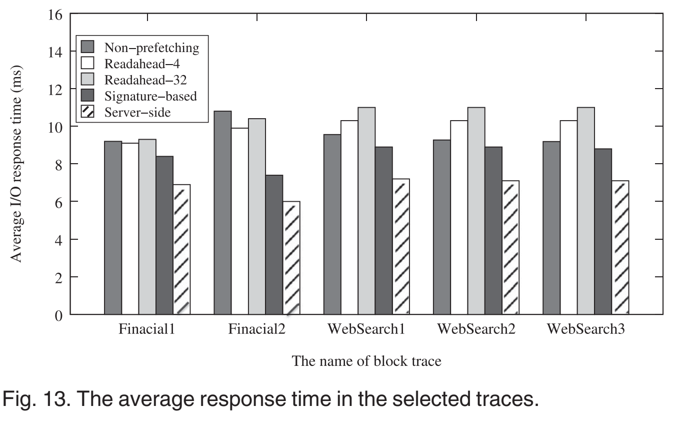


Fig. 13. The average response time in the selected traces.

prefetching block data. In our proposed mechanism, client identification is piggybacked to I/O requests and on the server side the offset sequence of block access events is transformed into a horizontal visibility graph. We translate a time series of block access events into a connected graph and classify block access patterns by employing the *Tarjan* algorithm, a graph theory technique to find cut vertices in a connected graph. Furthermore, we represent access patterns in feature tuples in order to save memory space and to conduct pattern matching quickly and accurately. The experimental results with micro-benchmarks and real-system block traces have shown that our server-side prefetching mechanism can reduce I/O response time and improve read data throughput compared to *Non-prefetching*, *Read-ahead* prefetching and *Signature-based* prefetching. We have also confirmed that both time and space overhead of the newly proposed mechanism are acceptable, as all analysis is moved to the storage server side. We also emphasize, that the idea of storage server-side data prefetching presented in this paper can be naturally applied to other conventional distributed/parallel file systems such as Lustre, the Google file system, PVFS or the Hadoop distributed file system.

Our current implementation of the prefetching mechanism requires the client file system to contact the metadata server for obtaining information on storage servers regardless of the cached data is hit or not. In the future, we intend to optimize PARTE's implementation of the client file system so that it examines buffered data before communicating to the metadata server. Moreover, client file systems do not currently piggyback any information regarding which pieces of the prefetched data blocks have been hit, and thus, the storage servers have no knowledge on the effectiveness of the prefetching mechanism. In the future, we are planing to piggyback cache hit information on the I/O requests of the client file system, so that storage servers can keep track of the quality of service and guide further optimization adjustments on their prefetching polices.

Acknowledgments

This work was partially supported by "National Natural Science Foundation of China (No. 61303038 & No. 61303227)", "Natural Science Foundation Project of CQ CSTC (No. CSTC2013JCYJA40050)", "the Scientific Research Foundation for the Returned Overseas Chinese Scholars, State Education Ministry", and "the Opening Project of State Key Laboratory for Novel Software Technology (No. KFKT2014B17)". The authors would like to thank anonymous reviewers for their thorough reviews and highly appreciate the comments and suggestions, which significantly contributed to revise this paper.

References

[1] (2014). HDFS: Hadoop distributed file system [Online]. Available: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

[2] (2012). IOSIG+ software [Online]. Available: https://code.google.com/p/iosig/

[3] (2013). SysBench benchmark [Online]. Available: http://sysbench.sourceforge.net

[4] (2013). IOzone filesystem benchmark [Online]. Available: http://www.iozone.org

[5] (2013). MySQL database server [Online]. Available: http://dev.mysql.com/

[6] (2010). Filesystem in userspace [Online]. Available: http://fuse.sourceforge.net/

[7] (2015, Apr. 4). UMass trace repository: OLTP application I/O and search engine I/O [Online]. Available: http://traces.cs.umass.edu/index.php/Storage/Storage

[8] S. Wu, F. Wang, X. Shi, H. Jin, et al., "Network I/O load based virtual machine placement algorithm in HPC cloud," *Sci. China Inf. Sci.*, vol. 42, no. 3, pp. 290–302, 2012.

[9] H. Lei and D. Duchamp, "An analytical approach to file prefetching," in *Proc. USENIX Annu. Tech. Conf.*, 1997, p. 21.

[10] E. Shriver, C. Small, and K. Smith, "Why does file system prefetching work?," in *Proc. USENIX Annu. Tech. Conf.*, 1999, p. 6.

[11] J. Liao, L. Li, H. Chen, et al., "Adaptive replica synchronization for distributed file systems," *IEEE Syst. J.*, vol. 9, no. 3, pp. 865–877, Sep. 2015.

[12] P. J. Denning, "The locality principle," *Commun. ACM*, vol. 48, no. 7, pp. 19–24, 2005.

[13] H. M. Song, Y. Yin, X. H. Sun, R. Thakur, et al., "Server-side I/O coordination for parallel file systems," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2011, pp. 1–11.

[14] B. Luque, L. Lacasa, F. Ballesteros, and J. Luque, "Horizontal visibility graphs: Exact results for random time series," *Phys. Rev.*, vol. 80, no. 4, article 046103, 2009.

[15] J. Liao, "Self-tuning optimization on storage servers in parallel file system," *J Circuits, Syst. Comput.*, vol. 30, no. 4, p. 21, 2014.

[16] S. P. Vanderwiel and D. J. Lilja, "Data prefetch mechanisms," *ACM Comput. Surveys*, vol. 32, no. 2, pp. 174–199, 2000.

[17] Y. Yin, S. Byna, H. Song, X. Sun, and R. Thakur, "Boosting application-specific parallel I/O optimization using IOSIG," in *Proc. IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput.*, 2012, pp. 196–203.

[18] P. Lu and K. Shen, "Multi-layer event trace analysis for parallel I/O performance tuning," in *Proc. Int. Conf. Parallel Process.*, 2007, pp. 12–21.

[19] A. Konwinski, J. Bent, J. Nunez, and M. Quist, "Towards an I/O tracing framework taxonomy," in *Proc. 2nd Int. Workshop Petascale Data Storage*, 2007, pp. 56–62.

[20] X. Ding, S. Jiang, F. Chen, K. Davis, and X. Zhang, "DiskSeen: Exploiting disk layout and access history to enhance I/O prefetch," in *Proc. USENIX Annu. Tech. Conf.*, 2007, p. 261–274.

[21] S. Jiang, X. Ding, Y. Xu, and K. Davis, "A prefetching scheme exploiting both data layout and access history on disk," *ACM Trans. Storage*, article 10, vol. 9, no. 3, 2013.

[22] K. Vijayakumar, F. Mueller, X. Ma, and P. Roth, "Scalable I/O tracing and analysis," in *Proc. 4th Annu. Workshop Petascale Data Storage*, 2009, pp. 26–31.

[23] S. Byna, Y. Chen, X. Sun, and W. Gropp, et al., "Parallel I/O prefetching using MPI file caching and I/O signatures," in *Proc. ACM/IEEE Conf. Supercomput.*, 2008, pp. 1–12.

[24] T. Madhyastha and D. Reed, "Learning to classify parallel input/output access patterns," *IEEE Trans. Parallel Distrib. Syst.*, 2002, pp. 802–813.

[25] J. He, J. Bent, A. Torres, and X. Sun, "I/O acceleration with pattern detection," in *Proc. 22nd Int. ACM Symp. High Perform. Parallel Distrib. Comput.*, 2013, pp. 26–35.

[26] J. Lewis, M. Alghamdi, M. Al Assaf, and X. Qin, "An automatic prefetching and caching system," in *Proc. 29th Int. Perform. Comput. Commun. Conf.*, 2010, pp. 180–187.

[27] R. Patterson, G. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed prefetching and caching," in *Proc. 15th ACM Symp. Operat. Syst. Principles*, 1995, pp. 79–95.

[28] M. Al Assaf, X. Jiang, M. Abid, and X. Qin, "Eco-Storage: A hybrid storage system with energy-efficient informed prefetching," *J. Signal Process. Syst.*, vol. 72, no. 3, pp. 165–180, 2013.

[29] I. Zhang, "Efficient file distribution in a flexible," Ph.D. dissertation, Massachusetts Inst. Technol., Cambridge, MA, USA, 2009.

[30] J. Stribling, Y. Sovran, et al., "Flexible, wide-area storage for distributed systems with WheelFS," in *Proc. 6th USENIX Symp. Netw. Syst. Des. Implementation*, 2009, pp. 43–58.

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

[32] V. Padmanabhan and J. Mogul, "Using predictive prefetching to improve world wide web latency," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 26, no. 3, pp. 22–36, 1996.

[33] Z. Li, Z. Chen, S. Srinivasan, and Y. Zhou, "C-Miner: Mining block correlations in storage systems," in *Proc. 3rd USENIX Conf. File Storage Technol.*, 2004, pp. 173–186.

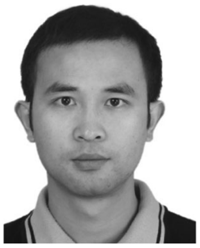
[34] N. Tran and D. Reed, "Automatic ARIMA time series modeling for adaptive I/O prefetching," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 4, pp. 362–377, Apr. 2004.

[35] R. Tarjan and U. Vishkin, "An efficient parallel biconnectivity algorithm," *SIAM J. Comput.*, vol. 14, no. 4, pp. 862–874, 1985.

[36] G. Soundararajan, M. Mihailescu, and C. Amza, "Context-aware prefetching at the storage server," in *Proc. USENIX Annu. Tech. Conf.*, 2008, pp. 377–390.

[37] S. Liang, J. Song, and X. Zhang, "STEP: Sequentiality and thrashing detection based prefetching to improve performance of networked storage servers," in *Proc. IEEE 27th Int. Conf. Distrib. Comput. Syst.*, 2007, p. 64.

[38] D. Pompili, "Uncertainty-aware autonomic resource provisioning for mobile cloud computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 8, pp. 2363–2372, Aug. 2015.



Jianwei Liao received the MS degree in computer science from the University of Electronic Science and Technology, China, in 2006, and then received the PhD degree in computer science from the University of Tokyo, Japan, in 2012. Now, he works for the college of computer and information science, Southwest University, China. His research interests are dependable operating systems and high performance storage systems for distributed computing environments.

François Trahay received the MS and PhD degrees in computer science from the University of Bordeaux, France, in 2006 and 2009, respectively. He is currently working as an associate professor at Telecom SudParis, France. He was a postdoc researcher at the RIKEN Institute in the Ishikawa Lab, University of Tokyo, in 2010. His research interests include runtime systems for high performance computing (HPC) and performance analysis.

Balazs Gerofi received the MS and PhD degrees in computer science from the Vrije Universiteit Amsterdam, in 2006 and The University of Tokyo, in 2012, respectively. He is currently working as a research scientist in the RIKEN Advanced Institute for Computational Science. His research is mainly focused on operating systems, high-performance computing, cloud-computing, and fault tolerant computing. He is a member of the *IEEE Computer Society and the Association for Computing Machinery (ACM)*.

Yutaka Ishikawa received the BE, MTech and PhD degrees in electrical engineering from the Keio University, Japan, in 1982, 1984, and 1987, respectively. He is a professor at the Department of Computer Science, the University of Tokyo, Japan. He was a visiting scientist in the School of Computer Science, Carnegie Mellon University from 1988 to 1989. His current research interests include parallel/distributed systems and dependable system software. He is a member of the IEEE.