IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received October 28, 2016, accepted December 17, 2016, date of publication December 21, 2016, date of current version January 27, 2017.

Digital Object Identifier 10.1109/ACCESS.2016.2642918

Optimization of Reading Data via Classified Block Access Patterns in File Systems

JIANWEI LIAO[1,2] AND SHANXIONG CHEN[1]

[1]College of Computer and Information Science, Southwest University, Chongqing 400715, China
[2]State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China

Corresponding author: J. Liao (liaotoad@gmail.com)

This work was supported in part by the National Natural Science Foundation of China under Grant 61303038 and Grant 61303227 and in part by the Opening Project of State Key Laboratory for Novel Software Technology under Grant KFKT2016B05.

ABSTRACT This paper proposes a novel mechanism to model a time series of block accesses for profiling block access patterns, to intentionally direct block data prefetching. The basic idea behind this scheme is that the block accesses within a certain offset domain may have some correlations that may contribute to classify an access pattern on the file system level. Moreover, the technique of adjacency matrix is employed to represent an access pattern for accelerating pattern matching, and then benefits I/O optimization eventually. Through a series of emulation experiments based on several realistic block traces on the disk, the experimental results show that the newly proposed prefetching mechanism outperforms other comparisons. Specifically, it can reduce average I/O response time by 14.6%–17.9% in contrast to the commonly used sequential prefetching scheme, and 4.1%–10.5% compared with frequent sequence mining-based prefetching but with less space and time overhead.

INDEX TERMS Block access pattern, data prefetching, I/O optimization.

I. INTRODUCTION

As the speed difference between processors and disks is becoming larger, the requirements for high performance storage systems in data-intensive applications are significantly increasing recently [3]. Data prefetching is a widely adopted I/O optimization technique for boosting disk performance, which speculative reads the data from disk in advance, on the basis of the predicted future I/O requests. Especially, disk-level prefetching schemes have been proposed recently, which have to forecast the future block accesses on the disk based on the analysis of occurred accesses [9], [15]. As a consequence, they can properly prefetch block data to mask disk service time from the view point of application workloads.

In fact, Chen *et al.* [12] have verified that blocks reveal semantic relationship, which has great scientific values and indicates a promising prospect in applications. Generally, disk-level prefetching can directly utilize the information about block layout and the correlations among blocks to forecast the future access events. It then can contribute to achieve better I/O performance, in contrast to the prefetching mechanisms focusing on the logical file level [9]. For instance, *C-miner* is a typical scheme to disclose the correlations among blocks for generating frequent sequences, and then advances forecasting future accesses to dominate I/O optimization [15]. Similarly, S. Jiang et al. have implemented *DiskSeen*, which employs a frequent sequence-based pattern modeling technique to classify block access patterns, 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 [9], [11]. Similarly, frequent sequence-based data mining schemes are commonly adopted by researchers working in different study areas, such as the work presented by Farzanyar and Mohammadreza [14]. They have proposed a local algorithm for tracing frequent item sets over a P2P network, for a possible future analysis.

However, the existing disk-level prefetching mechanisms including *C-miner* and *DiskSeen*, normally employ frequent sequence mining-based association rules to foresee the future block accesses. This kind of prediction approach has two limitations: 1) there are a huge number of fixed frequent sequences, thus, it is a time-consuming task to find a matched one for the current block accesses in the prediction window [14]; 2) it is possible to discover the access patterns with certain frequency on the same blocks, but it cannot disclose the access patterns targeting to different blocks, even if they might have same correlations, such as same offset difference between neighbor accesses.

To overcome the aforementioned shortcomings in the frequent sequence-based prediction approach for directing data prefetching, this paper proposes a domain-based scheme to classify block access patterns. Moreover, we introduce an adjacency matrix-based pattern matching algorithm to locate a matched pattern for the block accesses in the prediction window, with acceptable time and space overhead. To put it from another angle, we first transform a block access sequence to an undirected graph using the horizontal visibility graph technique (HVG) [1]. After that, we employ certain advanced (modified) graph algorithms to classify block access patterns according to the reference of locality, as well as to find matched patterns, for eventually instructing the process of data prefetching. In brief, this paper makes the following two contributions:

- *Employing HVG to pre-process the block access sequence.* We utilize HVG to transform a block access sequence, in which each access event has its relevant logical block address, to an undirected connected graph. As a consequence, we can make use of existing algorithms in graph theory to process the connected graph for completing the task of modeling access patterns. For instance, we can use the *Tarjan* algorithm for discovering all cut verties in the graph to preliminarily generate access patterns.
- *Proposing the adjacency matrix-based pattern matching algorithm.* To reduce the time required for seeking out the matched pattern, we have proposed a matrix-based pattern matching algorithm. In this algorithm, both fixed access patterns and the pattern of access events in the prediction window are represented with relevant adjacency matrices. That is to say a round of pattern matching can be satisfied by making a comparison between two matrices.

The rest of paper is structured as follows: the specifications of modeling block access patterns and performing pattern matching are presented in Section II. Section III describes the evaluation methodology and reports relevant results. Finally, we conclude the paper in Section IV.

II. PATTERN MODELING AND MATCHING

This section introduces the mechanism of modeling access patterns by analyzing the sequence of occurred block accesses, as well as the algorithm of pattern matching, both of them can definitely conduce to data prefetching.

A. MODELING BLOCK ACCESSES

A sequence of block access events can be measured at successive events that appear at uniform time intervals. Thus, this kind of sequence is commonly regarded as a typical time series [7]. Consequently, for the purpose of exploring the correlations of block accesses with a certain offset domain, we have employed the horizontal visibility graph technique, to pre-process the block access sequence. The technique of HVG was proposed to generate a mapping network from a given time series [1]. That indicates HVG can transform a sequence of occurred block accesses to a connected graph. To be specific, a data point in the sequence of block accesses can be simply expressed as $Access(time, offset)$, and we only focus on the offsets of the block accesses when modeling and classifying patterns. Actually, two nodes in the horizontal visibility graph are connected if one can draw a horizontal line in the time series by complying with the connection rule, which is explicitly defined as following:

Connection Rule: When $Access(t_x, off_x)$ *is connected with* $Access(t_z, off_z)$*, for an arbitrary* $Access(t_y, off_y)$*, if* $t_x < t_y < t_z$*, it must have* $off_x > off_y$ *and* $off_z > off_y$.

Figure 1(a) demonstrates an example of a horizontal visibility graph that is transformed from a sample time series of block accesses. In the figure, the bar of each access event indicates the offset of block, i.e. logical block address. In fact, we have demonstrated that *HVG* can facilitate classifying block access patterns [4].

Because we intend to use advanced algorithms in graph theory to classify access patterns, as well as perform pattern matching to benefit data prefetching, we transform the horizontal visibility graph to an undirected connected graph. Specifically, we first represent every bar in the HVG graph as a node in the connected graph. Then we link two nodes with an undirected edge, `if-and-only-if` their corresponding bars are also connected in the HVG graph. Figure 1(b) illustrates an example of undirected connected graph, which is corresponding to the HVG graph shown in Figure 1(a). Both figures contain same information about node connection, i.e. the connection relationship among block access events.

B. CLASSIFYING ACCESS PATTERNS

We discussed before, an undirected connected graph is corresponded to a horizontal visibility graph, which was originally generated from a history of occurred block access events. By referring back to Figure 1(b), the located cut points can separate the connected graph into several sub-graphs. Each sub-graph can be regarded as a potential access pattern, when the number of nodes in the sub-graph is in the reasonable given range by the reference of locality. Therefore, for the purpose of effectively disclosing the cut vertices in the connected graph, we utilize a modified *Tarjan* algorithm [8] to complete this task, and then explore fixed block access patterns from block access traces.

As seen, Figure 2(a) is a sub-figure of the connected graph of Figure 1(b), and can be easily transformed to another form of connected figure, which is illustrated in Figure 2(b). After collecting the fixed block access patterns representing with undirected connected figures through analyzing access traces, it is necessary to consider how to store them, and then how to use them for efficiently conducting pattern matching. In this paper, we use adjacency matrices to represent the fixed access patterns, because we aim to minimize the time needed for pattern matching.

To be specific, for representing an undirected connected graph with an adjacency matrix, we use "1" to indicate that two nodes are connected, and vice versa. The value of the entry in the 3rd row and 4th column of the matrix

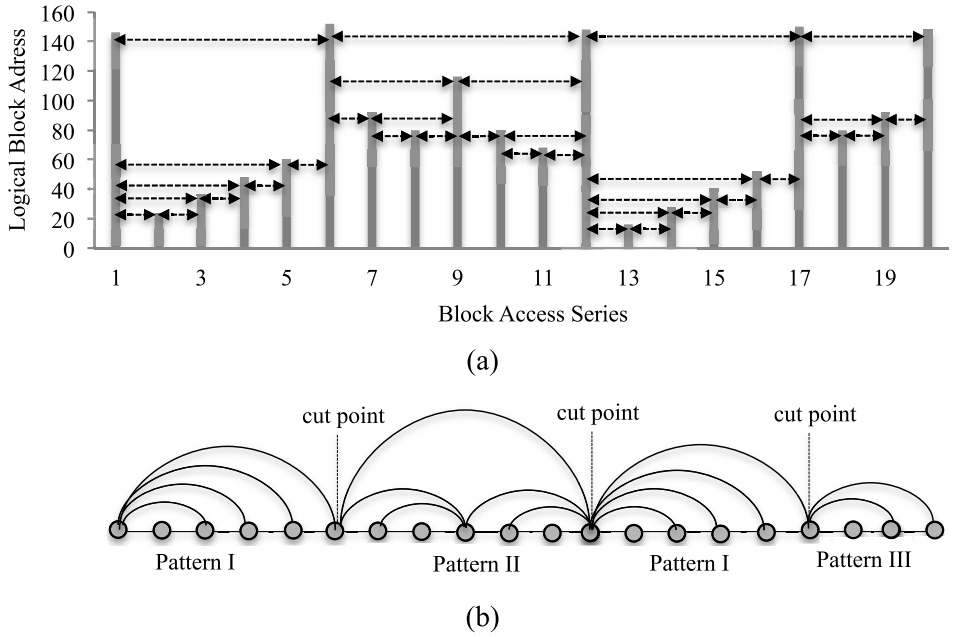

FIGURE 1. Modeling blocks access events by using *HVG*. (a) Horizontal visibility graph of block accesses. (b) Access pattern sub-graphs.

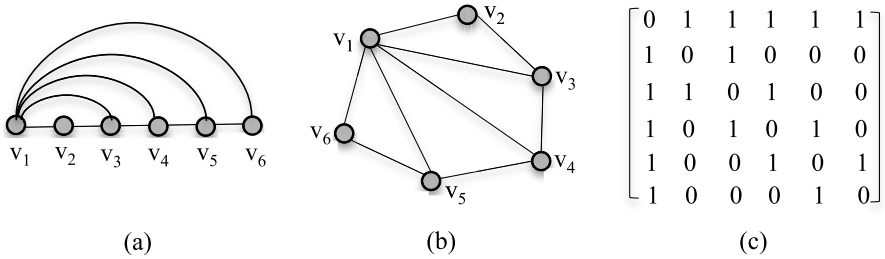

FIGURE 2. Classifying an access pattern and representing it with a adjacency matrix. (a) Access pattern I. (b) Connected graph. (c) Adjacency matrix.

shown in Figure 2(c) is 1, which means *No. 3 Access Event* (the third occurred access event) and *No. 4 Access Event* (the fourth occurred access event) in *Access pattern I* (shown in Figure 2(b)) are connected. Otherwise, the value of matrix element is set as 0 by default, if there is no connection between two relevant access events. Figures 2(b) and 2(c) demonstrate how to denote a collected (fixed) access pattern, i.e. *Access pattern I* by taking advantage of an undirected graph and a relevant adjacency matrix respectively. Moreover, all adjacency matrices of collected block access patterns are saved in a linked list, and the patterns having the same number of access events are organized together in a group.

C. PATTERN MATCHING ALGORITHM

In general, data prefetching is triggered when a matched pattern has been found for the current block accesses. This is because the information (offsets and sizes) about future block accesses can be retrieved in advance, by resorting to the matched access pattern. We present the algorithm of pattern matching in details in the section.

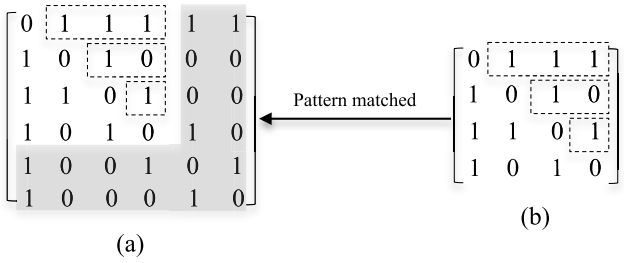

FIGURE 3. Pattern matching with adjacency matrices (note: $Number_{predict}$ is 2, and $Number_{window}$ is 4 in this example). (a) The fixed pattern. (b) Current pattern in prediction window.

Figure 3 demonstrates the main idea of the newly proposed algorithm for pattern matching. More specifically, a round of pattern matching in our proposed mechanism is conducted by the following steps:

1) We first build a corresponding adjacency matrix for the block access events in the prediction window. The prediction window contains the block accesses occurred much lately (i.e. current ones), and the following ones are needed to be prognosticated.

2) According to the number of access events requiring to be forecast, we try to find all fixed patterns that have totally $Number_{predict} + Number_{window}$ access events. In which the variable of $Number_{predict}$ is the number of events that are required to be forecast, and the variable of $Number_{window}$ indicates the number of occurred block access events that will be used for performing prediction. In other words, the number of block access events in these located (fixed) patterns is the sum of the number of accesses to be predicted and the number of accesses in the prediction window.
3) For each fixed pattern having required number of block access events, only the values in the matrix related to the previous $Number_{window}$ access events are supposed to be compared with relevant access events in the current prediction window. We conduct a right shift of $Number_{predict}$ bit positions on each row in the matrix of the fixed pattern, and compare the corresponding elements in two matrices. If there is an unmatched element, we proceed with checking the subsequent ones in the same way.
4) Finally, the algorithm will finish only if the match is found or all fixed patterns that have the required number of block access events have been traversed.

We have also studied the time complexity of our pattern matching algorithm: since it requires to traverse all access patterns have the expected number of involved access events. Consider that we have a total of M collected 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. Generally, it is necessary to set a reasonable range for N, to make the matching algorithm working more efficiently.

D. PREDICTION AND PREFETCHING

The main purpose of using adjacency matrices to express the connection relations among occurred block accesses, is to accelerate the speed of pattern matching. Once the matched pattern has been located, its raw data about offset differences and sizes of access events are supposed to be used to foresee the logical block address and size of the future block access. Namely, the logical block address of the future access can be computed by considering real offset differences illustrated in both the matched pattern and the accesses in the prediction window. Similarly, the size of the future access can be obtained with the same way as well.

After understanding the logical block addresses and the sizes of future block accesses, the storage system can read the required data in advance, to mask the time overhead resulted from heavy disk operations. Consequently, the prefetched data are cached in the file local cache, to immediately respond to application I/O requests, when the predictions hit.

In summary, the newly proposed forecasting scheme is rather different from the commonly used frequent sequence-based prediction schemes. It abstracts connection relationship of block access events from block access patterns to group these patterns. As a result, it not only reduces storage space for saving the fixed patterns, but also significantly accelerates pattern matching.

III. EXPERIMENTS AND EVALUATION

This section conducts evaluation to show strongpoints of the newly proposed approaches for classifying and matching block access patterns, to finally support block data prefetching. We utilized the evaluation methodology described in [15], which suggests to do trace-driven simulations with several disk traces collected in real systems. Therefore, we constructed a experimental platform by taking advantage of a widely used the *DiskSim* simulator [2], with a storage cache simulator, i.e. *CacheSim* [5], to emulate a real storage system that has totally 16MB storage cache. Besides, the Least Recently Used (LRU) replacement policy implemented by *CacheSim* has been leveraged to evict and load the prefetched data.

We selected several disk traces for evaluation. A block I/O traces with a one-week period span starting from 5PM GMT on 22nd February 2007 was selected as another benchmark, i.e. the *MSRC* trace. This trace collection is offered by 13 enterprise servers for different applications including web/SQL server, media server and web proxy. Therefore, *MSRC* is universally identified as the representative trace to cover all major access patterns [6], [13]. Since prefetching schemes make sense for read requests, we have only selected some read-intensive workloads from the *MSRC* trace. Furthermore, *TPC-C benchmark*, which can simulate a typical online transaction processing (OLTP) workload [10] was used in our simulations, as well.

Apart from our proposed *Domain*-based prefetching approach, the following three comparison counterparts have been used in experiments:

- *Non-prefetching*, which means the distributed file system without any I/O optimization facilities, such as prefetching. Consequently, there is no overhead introduced by analyzing block accesses and prefetching data. It has been selected as our *Baseline* for other comparisons, to demonstrate the gain/loss in a global scale resulted by other comparisons.
- *Sequential* prefetching, which is also called the *Readadhead* scheme. Namely, when there are non-consecutive misses to the server, the file system is supposed to issue a prefetch request reading the data of certain consecutive blocks in advance.
  Different from our proposal, however, the *Sequential* prefetching scheme fixedly fetches the data of following blocks (after the target block according to the current I/O request). That implies that the rule used to guide prefetching data is unchanged in all cases, though there will be plenty of stride read requests.
- *Frequent* sequence-based prefetching, which was firstly implemented by *C-miner*, and it explores the frequency sequences to conduce to forecasting future block accesses [15]. As a matter of fact, the *Frequent*

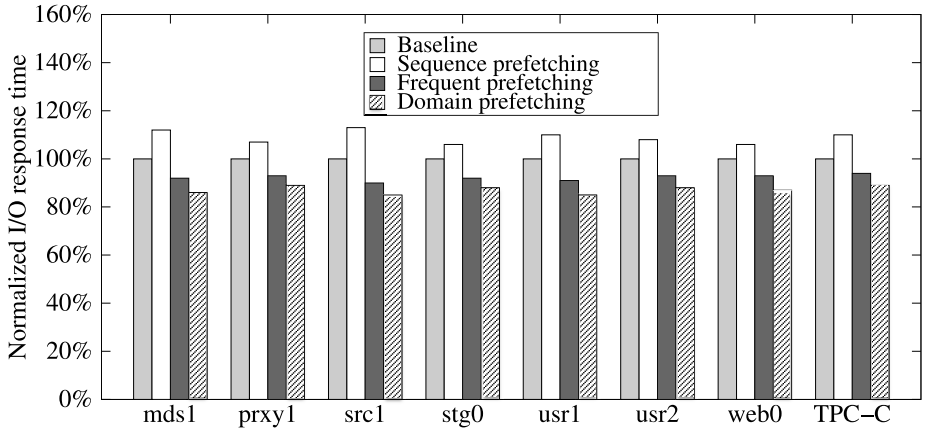

FIGURE 4. The normalized I/O response time with various prefetching schemes.

sequence mining based prefetching approach is heuristic and effective for different read patterns, it is the most related work to our newly proposed mechanism. As discussed before, however, this approach has its own shortcomings and may fail to work effectively for general cases of block accesses.

Considering block access patterns are supposed to be updated from time to time, we divided the selected trace of block accesses into several pieces, and each part may have 20, 000 access events. Then, for each part of the trace, the first 25% trace of block accesses are used to classify access patterns, that is to say the prefetching works for the remaining 75% accesses. The number of access events in each fixed access pattern was constrained to the interval of [4, 12]. Besides, we have conducted the experiments to measure the effects on performance with *Sequential* prefetching by reading different size of block data in advance, but we have disclosed that there are no much difference while prefetching data of 2, 4 or 6 blocks, so that we set the number of prefetched block data as 2 while employing *Sequential* prefetching. Actually, with the exception of *Non-prefetching*, other three prefetching schemes have 4MB prefetching cache, which is a part of the storage cache, for saving the prefetched data. The following sub-sections explore both positive and negative aspects of this newly proposed mechanism respectively.

Before evaluating the proposed prefetching mechanism, we have analyzed characteristics of disk block accesses in different workloads. Table 1 shows the details of selected workloads used in our simulations. All of them are abstracted from two standard benchmark suits, i.e. the *MSRC* trace and TPCC, reflecting high-end enterprise environments.

A. I/O RESPONSE TIME

We first measured the I/O response time in the emulated storage systems, which may equip with different prefetching schemes. Since the response time varies in size from each other greatly while running different disk traces in

TABLE 1. Characteristics of selected workloads.

Traces	Read Ratio	Size per Read	Description
mds1	93%	56.8 KB	Media server 1 at MSRC
prxy1	65%	12.6 KB	Web proxy 1 at MSRC
src1	98%	59.3 KB	Source control 1 at MSRC
stg0	64%	40.8 KB	Web staging 0 at MSRC
usr1	91%	49.5 KB	User home dirs 1 at MSRC
usr2	81%	43.8 KB	User home dirs 2 at MSRC
web0	99%	10.2 KB	Web search 0 at MSRC
TPC-C	67%	8.2 KB	Microsoft TPC-C (OLTP)

experiments, we show the normalized response time in Figure 4. In the figure, X-axis shows the name of disk traces, and Y-axis illustrates the normalized I/O response time (the lower one is better).

Clearly, the *Sequential* prefetching scheme achieves the worst I/O response time, that is to say *Sequential* prefetching may place negative effects on the storage systems when the major part of the disk traces are not sequential. Compared with other comparison prefetching schemes, our newly proposed *Domain*-based prefetching mechanism can reduce average I/O response time by $14.6-17.9\%$ in contrast to *Sequential*, and $4.1-10.5\%$ compared to *Frequent*. For instance, *Domain* can save more than 6.1% response time than *Frequent*, and 13.2% time than the selected *Baseline*, when the disk trace is *web0 at MSRC*. In summary, the proposed mechanism outperforms others in the metrics of I/O response time, it thus can speedup the executions of applications.

B. PREDICTION ANALYSIS

For showing the feasibility of our proposed prediction model in practice, there is a need to check the prediction error/deviation when applying different estimation models on the varied workloads in the selected benchmarks. This section intends to report the statistics relating to read predictions and prediction hits, by employing different estimating models. As *Baseline* does not perform read prediction and data prefetching, there are no relevant prediction statistics, so that

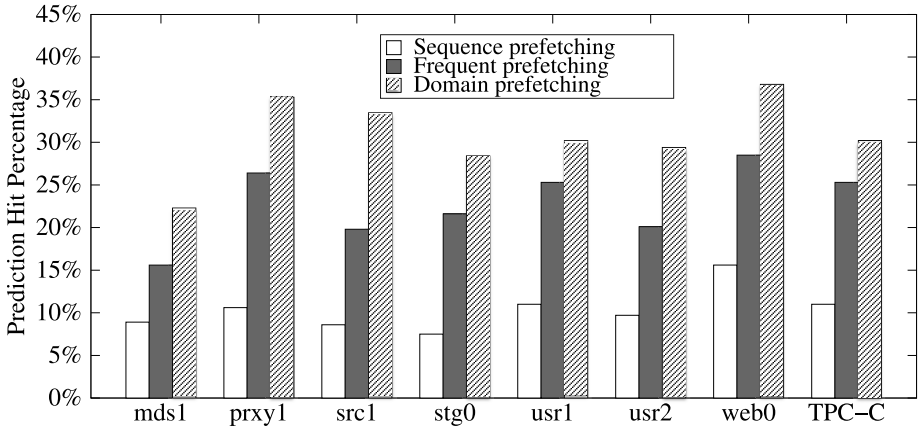

FIGURE 5. Prediction analysis with different prediction models adopted by different prefetching schemes.

TABLE 2. Time overhead for various prefetching schemes (seconds).

Disk Traces	Baseline (non-prefetching)	Sequential (prefetching)	Frequent (modeling+matching+prefetching)	Domain (modeling+matching+prefetching)
mds1	0.0	10.3	$18.2 + 50.2 + 10.3$	$16.6 + 36.2 + 10.7$
prxy1	0.0	7.1	$22.3 + 69.5 + 10.1$	$18.6 + 43.6 + 9.8$
src1	0.0	11.1	$24.2 + 78.2 + 11.4$	$20.1 + 61.3 + 10.9$
stg0	0.0	8.5	$31.2 + 70.4 + 19.4$	$17.4 + 35.8 + 18.1$
usr1	0.0	9.7	$21.3 + 71.2 + 10.2$	$18.1 + 52.3 + 10.4$
usr2	0.0	10.3	$26.3 + 77.1 + 13.3$	$21.1 + 51.4 + 11.1$
web0	0.0	6.1	$24.3 + 59.3 + 8.8$	$18.1 + 49.8 + 8.9$
TPC-C	0.0	8.9	$44.3 + 131.6 + 22.1$	$22.8 + 55.8 + 20.3$

only the statistic data while using *Sequential*, *Frequent* and *Domain* prediction algorithms are presented.

Figure 5 shows the experimental results about prediction statistics by using different prediction models. In the figure, the horizontal axis indicates the name of workloads, and the vertical axis represents the percentage of prediction hits, which is the rate of (the number of prediction hits/total number of reads). The experimental results explicitly show that the proposed *Domain*-based prefetching mechanism can achieve more prediction hits by $19.2 - 24.7\%$ compared to *Sequential* prefetching mechanism, and $6.3 - 10.1\%$ compared to the commonly used *Frequent* sequence-based prefetching scheme. For example, the *Domain*-based prefetching approach can result in more than 8.3% prediction hits than *Frequent*, and 12.9% hits than the selected *Sequential*, when the disk trace is *web0 at MSRC*.

C. TIME AND SPACE OVERHEAD

After reporting the benefits resulted from the newly proposed prefetching mechanism, this section measures the overhead brought by various prefetching schemes. Besides reading block data in advance, predicting future block accesses definitely cause time overhead. This is because estimating future block accesses relies on analyzing block access history to generate fixed access patterns or sequences, as well as performing pattern matching.

As seen in Table 2, the split time overhead resulted from different operations while using various prefetching schemes, is explicitly reported. The *Baseline* scheme does not conduct prefetching, it does not result in any time overhead. And the *Sequential* prefetching approach indeed requires to read block data in advance; however, it performs neither modeling access patterns, nor matching patterns. On the other side, both *Domain* and *Frequent* schemes must to generate and manage the fixed access patterns or sequences, that is why both of them cause more time for completing data prefetching, compared with *Sequential*.

Another interesting clue shown Table 2 is that the newly proposed mechanism can reduce $27.7-50.0\%$ time overhead, compared with *Frequent*, though both of them require almost same time for reading the prefetched block data. In fact, the *Frequent* prefetching approach generates more fixed frequent sequences (the *modeling* part), and then spends more time to locate a matched frequent sequence for current block I/O events (the *matching* part). As a consequence, *Frequent* needs more time for completing data prefetching.

Furthermore, we recorded overall memory space for storing disk traces and the fixed access patterns, and Table 3 reports the results. Since both *Baseline* and *Sequential* do not buffer logs of block access events for classifying access patterns, they do not cause any space overhead. But, the presented *Domain*-based prefetching scheme is efficient in terms

TABLE 3. Space overhead for various prefetching schemes (MB).

Traces	Baseline	Sequential	Frequent	Domain
mds1	0.0	0.0	79.3	71.9
prxy1	0.0	0.0	171.1	161.6
src1	0.0	0.0	101.4	90.8
stg0	0.0	0.0	99.8	84.0
usr1	0.0	0.0	129.3	112.1
usr2	0.0	0.0	132.8	122.9
web0	0.0	0.0	91.7	78.6
TPC-C	0.0	0.0	166.2	138.7

of space overhead, in contrast to *Frequent*. More exactly, our proposal can save 6.3−17.1% memory space for guiding data prefetching.

IV. CONCLUDING REAMARKS

This paper has proposed and evaluated a *Domain*-based prefetching mechanism, which classifies block access patterns occurred on disks to finally conduce to data prefetching. Specifically, it employs the horizontal visibility graph technique to model the history of block accesses, and then uses the *Tarjan's* algorithm to classify block access patterns. Moreover, the proposed mechanism employs an adjacency matrix-based pattern matching algorithm to speedup data prefetching. The experimental results demonstrate that our newly proposed mechanism can yield attractive I/O acceleration, and then significantly improve system performance with acceptable time and space overhead.

We have only evaluated the selected real-system workloads individually, as the current implementation fails to work for more complicated cases. In other words, this newly proposed mechanism cannot classify block access patterns while there are two or more workloads access disk blocks in parallel, because they might lead to interleaved block accesses. We are now in the process of sifting the block accesses belonging to the different workloads, to enable our proposal for supporting concurrent execution of more workloads.

ACKNOWLEDGMENT

The author would like to thank anonymous IEEE TC reviewers for their thorough reviews and highly appreciate the comments and suggestions, which significantly contributed to revise this paper.

REFERENCES

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

[2] G. R. Ganger, "System-oriented evaluation of I/O subsystem performance," Tech. Rep. CSE-TR-243-95, Dept. Comput. Sci. Eng., Univ. Michigan, Ann Arbor, MI, USA, Jun. 1995.

[3] J. Liao, F. Trahay, G. Xiao, L. Li, and Y. Ishikawa, "Performing initiative data prefetching in distributed file systems for cloud computing," *IEEE Trans. Cloud Comput.*, to be published, doi: 10.1109/TCC.2015.2417560.

[4] J. Liao, F. Trahay, B. Gerofi, and Y. Ishikawa, "Prefetching on storage servers through mining access patterns on blocks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 9, pp. 2698–2710, Sep. 2016.

[5] M. L. C. Cabeza, M. I. G. Clemente, and M. L. Rubio, "CacheSim: A cache simulator for teaching memory hierarchy behaviour," *ACM SIGCSE Bull.*, vol. 31, no. 3, p. 181, 1999.

[6] *MSR Cambridge Traces*, accessed on Feb. 20, 2014. [Online]. Available: http://iotta.snia.org/tracetypes/3

[7] N. Tran and D. A. 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.

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

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

[10] *TPC-C Database Benchmark Traces*, accessed on Dec. 2, 2013. [Online]. Available: http://tds.cs.byu.edu/tds/

[11] 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.*, San Francisco, CA, USA, 2007, Art. no. 20.

[12] Y. Chen, F. Li, B. Du, J. Fan, and Z. Deng, "A quantitative analysis on semantic relations of data blocks in storage systems," *J. Circuits, Syst. Comput.*, vol. 24, no. 8, p. 1550118, 2015.

[13] Y. Zhang *et al.*, "Warming up storage-level caches with bonfire," in *Proc. 11th USENIX Conf. File Storage Technol. (FAST)*, 2013, pp. 59–72.

[14] Z. Farzanyar and M. Kangavari, "Distributed frequent item sets mining over P2P networks," *Comput. Inform.*, vol. 34, no. 2, pp. 458–472, 2015.

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

JIANWEI LIAO received his Ph.D. degree in computer science from the University of Tokyo, Japan, in 2012. He joined the College of Computer and Information Science, Southwest University, China, in March, 2012. He is also with the State Key Laboratory for Novel Software Technology, Nanjing University, China. His research interests are system software and high performance storage systems for distributed computing environments.

SHANXIONG CHEN received the Ph.D. degree in computer science and technology from the College of Computer, ChongQing University. He was a Visiting Scholar with South Australia University in 2014. He is currently an Associate Professor with the College of Computer and Information Science, Southwest University, China. He currently holds the post-doctoral position with Southwest University. His research interests include machine learning, network security, and data mining.