ATLANTIS PRESS

Advances in Computer Science Research, volume 62

2nd Joint International Information Technology, Mechanical and Electronic Engineering Conference (JIMEC 2017)

Disk Prefetching Using Blocks' Association Rules

Minjie Sun

College Of Computer & Information Science, Southwest University, No.2 Tiansheng Road, Beibei Distract, Chongqing, 400715,P.R.China

Keywords: Association rule; prefetching; Bloom filter.

Abstract

Modern applications, especially I/O-bound workload such as Online Transaction Processing(OLTP) often suffer from the performance gap between processor and I/O devices. Too much time is wasted waiting for the I/O operations. And there seems little evidence that the gap will stop widening in the near future. In this paper, a novel disk prefetching scheme is proposed. This prefetcher utilizes the blocks' association rules that are recognized as semantic information embedded in disk I/O traces. The experimental results show that the proposed scheme can satisfy the demand for real-time response to I/O stream and the I/O performance can be effectively increased.

1 Introduction

As the speed gap between processor and disk continues to grow, the disk-based storage system is becoming the bottleneck of data-intensive application system. To satisfy the growing demand for high performance storage system, we are now allowed to apply much more delicate methods to the storage system to attack the problem in cost of integrating more advanced hardware to the storage systems.

Prefetching is one of the common solutions to boost the I/O performance. This approach hides the latency with early I/O initiation by more delicate prediction algorithm running on the relatively redundant computational resources.

In general, there are mainly two fundamental prefetching techniques : heuristic and speculative schemes. The later approach 'pre-executes' fragments of code of the main process via an extra helper thread. The speculative execution thread observes the I/O operation of the main process and identifies future I/O references whenever it consider properly. This pre-execution scheme generally achieves high accuracy and can be more suitable for applications whose access patterns are complex or random. However, it usually requires certain hardware support and source code transformation of the target application. Therefore, it cannot work on application whose source code is not available. Thus its application areas are limited by the inherited requirements.

On the other hand, the much less restricted heuristic scheme predicts future access based on patterns obtained by analyzing past access history (or so-called trace). Several studies try to build models using some classic mathematical analyses such as hidden Markov model and ARIMA time series model. Beside the aforementioned modeling approaches, other data mining techniques are also applied to the data prefetching. Most of the existing schemes are limited to scientific computing scenario, in which data access sequence presents a simple and regular pattern. As far as we know, the only industrialized prefetching mechanism deployed in general purpose operating system (Linux) is readahead algorithm, which only detects sequential reads on file-level and has no intention to get involved in other patterns.

In response to all these observations, we propose a novel association rule mining based prefetching approach. It is designed for improving the I/O performance for complex workloads. This scheme exploits the block correlations similar to Clospan, which utilizes the frequent sequence pattern mining technique called CloSpan. We derive a new mining approach, which solely focus on predicting future I/O request. It outputs the association rules that directly used to predict future data access. In order to satisfy the real-time response feature that any practical prefetcher should achieve, an optimized bloom filter based rule matching algorithm is given in the following sections.

This paper is organized as follows. In section 2, we describe the semantics of block correlations, and discuss the various properties and contains of association rule. The detailed mining and predicting algorithms are presented in section 3. Experiments and Evaluation are discussed in section 5. The conclusion is given in section 6.

2 Exploiting Block Correlations as Semantics

2.1 Block Correlations

It is a common sense that two blocks can be considered 'linked' if the data stored by these blocks is correlated. The essential concept of block correlations is based on this simple idea. The block correlations are widely distributed across the storage medium. Consider a source code file that is saved in sector n and sector n+1 of the disk. Then, we can recognize the association between these two sectors by the semantic sense the code delivery. However, data semantics can be much more complex and rich, thus making the block correlations hard or inefficient to be identified. Let us take the example of source code file again. As shown in Figure 1, there are three files, namely, x.h, y.h and z.c. We assume x.h and y.h is included by z.c and their corresponding sectors are distributed across the disk. These sectors are also related by C macro command #include, so the chance that they will be all read altogether is high. As a consequence, we can see that sectors with great spatial distance also have temporal closeness in the access stream during a compiling operation.

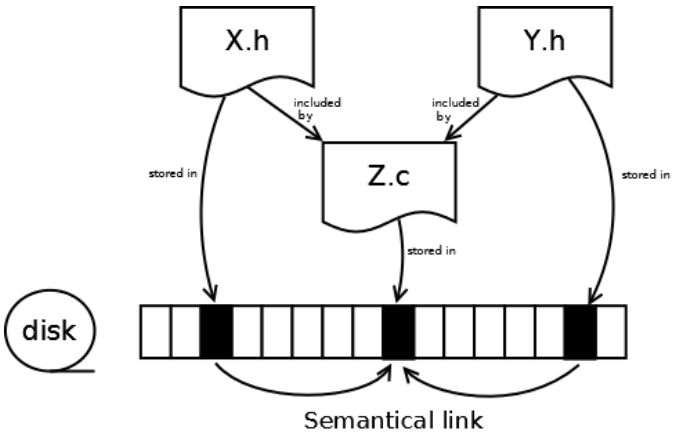


Figure 1: The semantical link among the three source code file

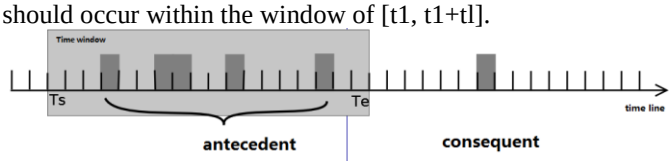
2.2 Association Rule with Time Constrains

Association rules describe a phenomenon that certain items occur with a high frequency after occurrences of other specific bunch of items. If we discover disk block 5 always requested after block 1, 2 and 3. Then we obtain one association rule represented as {1, 2, 3}->{5}. The association rules we use in this paper do not care about the order of the items' occurrences in the left hand of the implication expression, or called antecedent. This means the next time we observe the blocks in the order of {3,1,2}, the rule still applies. We only need mine one-item-consequent rules here. The term consequent refer to the right hand of implication.

To evaluate the effectiveness of all possible rules, various measurements that evaluate rules' significance and interest can be used. The two most common constraints are support and confidence. In general, rules with high support and confidence are usually picked up for further use. However, for predictive purpose, we should also consider the time constrains.

In order to restrict the search space, time constrains are applied during the mining stage. It is obvious that those rules are of less use even when they are indeed frequent occurrence and highly confident, if whose antecedents have a long time span or there is a huge time lag between the two antecedents and consequents. These two time constrains are shown in Figure 2

Formally, a time window w on a block sequence is represent as w[Ts, Te], where Ts and Te are the start time and the end time of window w. We usually constrain the width of the searching window so that only closely occurring sequences are searched. Besides, the consequent is only allowed to occur within a limit period of time right after antecedent stops, which means if the antecedent stops at time t1, and assume the maximum time lag is defined as tl, then the consequent should occur within the window of [t1, t1+tl].



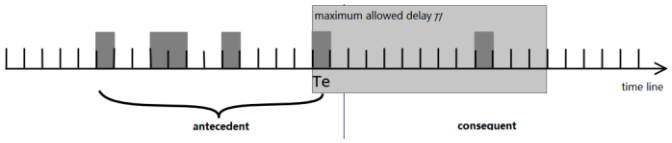


Figure 2: two time constains. (a)window on antecedent and (b)maximum time lag after antecedent.

3 Mining and Matching Rules

The mining algorithm proposed here use a pattern-growth approach to expend the antecedent with one item at a time. The main mining procedure consists of two main stages: 1) extracts one-item-antecedent frequent rules; 2) depth-first antecedent expending stage.

3.1 Association Rule with Time Constrains

The first mining procedure is to extract one-item antecedent rules. As an additional preprocess stage, the whole trace is cut up into subsequences in fixed length. The algorithm repeatedly scans each piece of the trace, records all qualified rules for each scan, and then merge the rules with those recorded in the previous scans. The following pseudo-code shows the algorithm.

Algorithm 1 one_item_mining(s, rsi)
Input: a piece of trace s, rule setrsi Output: rule set rso 1 rule set tmp_rs←{} 2 for each block reference b in s, generate time windoww [occur(b)+tl_min, occur(b)+tl+tl_max] 3 block set b1_set←{} 4 for each block reference b1 in w and not in b1_set 5 b1_set ← b1_set + b1 6 Callupdate(tmp_rs, rule(b->b1)) 7 end for 8 end for 9 filter out rules in tmp_rs whose support is lower than min_supp. 10 rso←merge(tmp_rs, rsi), rsi←rso.

The denotation occur(b) in line 2 represents the timestamp of block b, and update operation in line 6 adds a new rule b->b1 to tmp_rs if this rule is not found in tmp_rs, or updates rule's support and confidence if it is already in tmp_rs. Ideally, if we run the algorithm using the whole trace as one input, we will not fail to count those rules that happen to span across two pieces. However, for real system with limited capacity of memory, we have to apply divide and conquer approach. According to the analysis of …, the possibility of an instance of a frequent rule span over the cutting point is small, so the inherent loss is acceptable.

3.2 Association Rule with Time Constrains

Once one-item-antecedent rule set is obtained, prefetcher is able to work. However, the expansion stage is necessary. This is because of the observation that there are many rules sharing the same antecedent. Assuming rule {1,2,3}->10 and {1,3,5}->7 are hidden in the trace, we may get rules 1->10 and 1->7,

both of which can be used to guide prefetcher. If block 1 occurs in the stream, it either prefetches both block 10 and 7, or only selects the higher confidence block. Neither policy is suboptimal. In response to this issue, antecedent-expansion further mines the pattern, clears out the ambiguity of shorter rules, and directs the prefetcher making more confident and precise decisions. However, expansion is not always necessary when rules are already highly confident. So we skip the rules with confidence above 0.9.

The expending algorithm is as follows:

Algorithm 2 expand(*r*)

Input: rule *r* to be expanded
Output: rule set *rs*, return *1*,if rule *r* or one of more of its descendants saved. Return 0 otherwise.

```
1  rule set tmp_rs←{}
2  for each instance i of r
3     ts, te refer to timestamp of antecedent's start/stop, and
tc is timestamp of consequent's item
4     calculate searching window w[te-tw, min(ts+tw, tc)]
5   for each block b in w, such that b>max_bnr(r) and
rule(b->cons(r)) is
Included in rule set produced by last stage
6        Call update(tmp_rs, rule( {ante(r),b}->cons(r))
7     end for
8  end for
9   flag ←0
10for each candidate rule rc in tmp_rs such that
supp(rc)>min_support
11       if conf(rc) >cutoff_conf
12            put rc into rs
13flag ←1
14       else if conf(rc) >conf(r) and expand(rc) == 1
15        flag ←1
16       end if
17  end for
18  ifflag==0
19     if conf(r) >min_conf
20         put  r  into rs
21flag ← 1
22    end if
23end if
24 return flag
```

Operationupdate() does the same task as the one in previous stage and conf(r) denotes the confidence of rule r. The variable tw is pre-defined maximum searching time windows described in section 2.2. This algorithm consists of two parts. The first part collects all possible candidate blocks in a limited search space. On line 5, It only picks up blocks with greater numbers than those already included in rule r's antecedent so that it is guaranteed that no duplicate blocks are brought in. Another constrain is based on following lemma: if rule {a}->{b} is infrequent, none of the rules whose antecedents consists of item a and share the same consequent are frequent.

In the second part, the algorithm decides whether continue further searching and whether save current rule r. Only the candidates rules that have higher confidence than their ascendant but not higher than cutoff_conf(line 11) should be worth further mining, so that all rules left in rule set rs are at their highest confidence and searching space is pruned as another benefit. If there are no possible candidates worth mining, it checks current rule r's confidence and decides whether saving it. If either the rule r or any candidate has been saved (calling expand(rc) at line 14 returns 1), the algorithm returns 1 notifying it's upper caller that the candidate has been kept and there is no need to save the candidate's ascendant. This notification will pass all the way back to root, causing all the ascendants alongside the chain discarded, which is desired. Figure 3 shows the calling chain in different situations.

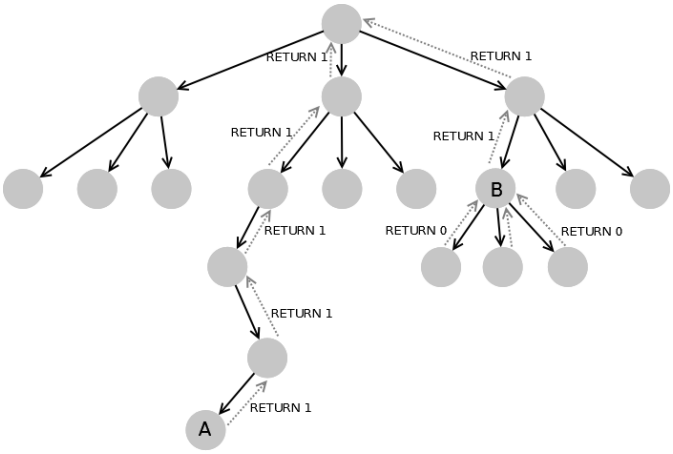


Figure 3: The calling chain in different situations

In Figure 3, the returns from lower calls. Search stops at node A and result is saved. The upper nodes return 1 all back to root. Node B's lower nodes all return 0, but its self meets the condition and saved , thus it return 1 back to its upper nodes.

Expanding the consequent using the algorithm above is not necessary. Instead, once antecedent expanding procedure is completed, the final step is grouping the rules by antecedent, and merging all the items of consequents of each group into one consequent to form the consolidated rule.

To accelerate to mining stage, portions of the algorithm can be parallelized. Taking algorithm 1 for example, iterations of loop of line 2 to 8 are independent on each other. Empowered by OpenMP , echo independent work can be assigned with a separated thread. Although, these worker threads can perform simultaneously on the multi-core system, some operations must be taken extra care of. For instance, update() operation among these thread needs to be performed in serial manner for the reckoning that it touches the shared data structure tmp_rs. Another issue is to avoid over-parallelizing, introduced by parallelizing the loop of line 10-17 of algorithm 2. Without extra caution, one thread will recursively fork into multiple threads on the call to expend() of line 14. In response, a threshold cutoff_level is defined to control the degree of parallelism: when depth of the search is greater than cutoff_level, the algorithm will call to single-thread version of the algorithm which means further searching is serially performed on the current thread. On the other hand, each search task is assigned with a thread if its depth is above cutoff_level. Figure 4 shows the idea of switching parallelized searching to serial searching:

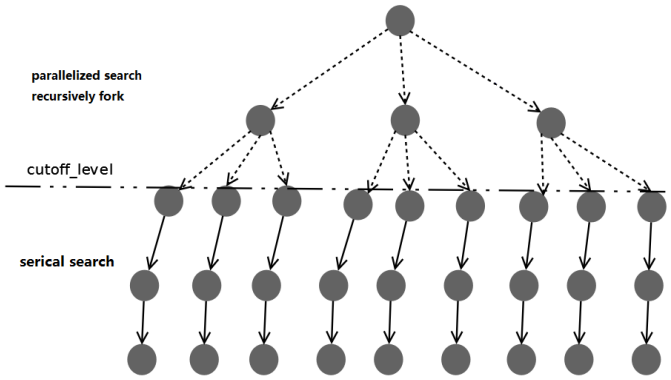


Figure 4 : the threshold cutoff_level

3.3 Rule matching

A rule set is obtained after the mining stage. The rules are queried to guide the prefetcher to make prediction of future I/O references. While the application is running, what the prefetcher can observe is a stream of request of blocks with time stamp attached to each item. For some applications, the stream is generated at a relatively high speed. Due to the limited space of memory, it is impossible to archive the entire data in memory. We are only allowed exploit limited number of items in the stream to match the rule set.

Observation window is defined as a range within which the items of stream are visible to the prefetcher. Ideally, the whole stream can fall into the observation window. But in practice, it is impossible and unnecessary. Instead, we define the observation window as the last few occurring items of the steam. And the width of window is set equal to that of search window described in section 3.2.

The rule set is stored in a key-value database, where antecedents are the keys and consequents being the value. To meet the requirement of the database that all keys should be unique, the items of antecedents are sorted before inserting to the database. From now on, we use term database to refer to the key-value database.

The baseline matching approach forms exhaustive combinations of the items within the observation window. And each combination is tried as the key to query the database.

Assume the sequence in the window is {a1, a2, .., an}. The key generation is as follows: 1) make one of the exhaustive combinations of first n-1 {a1,a2,..an-1} items; 2) concatenate the last item an to form each new sequences; 3) sort the sequences and inquiry the database with it. The matching is completed if any combination is found in the database or all the exhaustive combinations are tried, and no entry is found. Assume the sequence in observation window is {x,y,z,w}, the generated sequences are {z,w}, {y,w}, {x,w}, {z,y,w}, {z,x,w}, {y,x,w}, {z,y,x,w}. To make a success match as soon as possible, later items are prior to be selected for the combinations.

Consider the exponentially growth of combination generation, the observation window can only hold very limited items. Apart from that, the more serious issue is that the matching is very time-consuming because the inquiries are formed blindly.

Our improved matching algorithm incorporates bloom filter, which helps building informed inquiries. The number of hash functions is set equal to the maximum number of items in antecedents stored in database, and the number of elements is equal to that of rules. The number of bits of the bit array is determined by equations described in .

To set up the bit array, each rule is mapped onto a few specific bits on the bit array by the following steps: assume the antecedent being {a1, a2, …,an}, for each item a_k, 1) function i=g(a_k) maps to the actual hash function f_i ()and 2) f_i (a_k) maps a_konto the specific location of the array. These corresponding bits are then set to 1. The function g() is used to make sure the two hash functions used in inserting and querying are the same one. This meta-hash function g(a) here is simply defined as the reminder of the parameter a divided by the number of hash functions.

The matching algorithm is given in pseudo-code:

```
1    key ← {}
2    IF A[f(an)]  == 0
3            return RULE_NOT_EXSITED;
4    END IF
5    FOREACH a IN {a1, a2, …an}
6            IF A[f(a)] = 1
7                    Key ← {key, a}
8            END IF
9    END FOR
10   Sort key
11   WHILE key is not empty
12           IFsearch_in_database(key) == SUCCSESS
13                   return RULE_EXSITED
14           remove maximum item from key
15   END WHILE
16   return RULE_NOT_EXSITED
```

The algorithm consists of two parts. The first part, from line 1 to line 10, forms the possible key to be inquired. Notice line 2 and 3, matching will stop if the corresponding bit of last item of the window is zero. The second part inquiries the database with the key. Note that false positive key may be formed by the first part, although the possibility is low. Under this circumstance, meaning database can not retrieve the valid rule (the other branch on line 12), the algorithm remove the maximum item from the key,and then repeat the inquiry, till the key is empty.

In the worst situation, the algorithm tries to query for n times, while the baseline attempts up to 2ntimes. Bloom filter is the key data structure that avoids most false inquiries.

4 Evaluation

4.1 Methodology

To evaluate the performance of our approach, a simulation environment is implemented. It consists of a cache, various replacement policy, prefetcher and DiskSim. The environment simulates the functions performed by block I/O layer and disk driver of Linux kernel. When the system starts, it reads the disk trace as input, while maintaining the cache

and provokes replacement policy and prefetcher whenever it consider properly. And it is also implemented in the simulator. The actual disk access generated by the simulator is forwarded to DiskSim, which simulates handling request by real disk system.

We select the following traces for our experiments:

1. TPC-C trace[16] is collected on a storage system backing a Microsoft SQL Server. The trace is generated during the clients running TPC-C benchmark for 2 hours.
2. OLTP trace is collected by running OLTPBench, which simulates a various application's access to a database. In our configuration, OLTPBench simulates the behavior of TPC-C benchmark, and the database that handles the request is Mysql.
3. Kernel trace is collected by the authorsduring compiling Linux kernel. The compiling computer runs a modified Linux kernel, which intercepts VFS's routines and records every disk block references. The total time is about 48 minutes.

We only use the first half part of traces to mine association rules. And… evaluate the performance of prefetcher directed by the rules with the second part. We also implement sequential prefetching algorithm based on the same idea adapted by readahead[11] in Linux kernel. All experiments use Least Recent Used (LRU) cache replacement policy.

4.2 Overhead of Rule mining

The mining program is tested on a virtual machine with a 4-core processor and 8G memory. We measure the time the mining operation takes under different settings. Table 1 shows the results:

Trace	support	Time(sec.) Single-thread	Time(sec.) 4-thread	No. rule （K）
TPC-C	0.3	2240	580	160
	0.5	1350	350	95
	0.7	650	180	43
OLTP	0.3	1230	420	124
	0.5	890	240	89
	0.7	440	120	50
kernel	0.3	940	250	87
	0.5	790	200	64
	0.7	300	80	25

Table 1: Mining Time under different settings

As the table indicates, the parallelized algorithm significantly improves the speed then the original serial version. The former running time is about a quarter of the latter. Considering only the independent loop portion can be parallelized, and even in the parallelized portion there are operations that can only be performed serially. In addition to the thread scheduling overhead, the overall acceleration rate is always lower than 4.

4.3 Inquiries of rule matching

To evaluate the real-time feature of rule matching, we measure the average inquiries of both baseline matching and bloom filter based matching algorithms. The number is calculated as:

$$averageinquiries = \frac{totalinquries}{lengthofdisktrace}$$

Table 2 shows result:

Trace	Baseline approach	Bloom Filter based approach
TPC-C	15.75	0.17
OLTP	15.63	0.21
kernel	14.87	0.06

Table 2: Average inquiries

In our experiments, baseline matching set the observation window as last 5 item of the stream. As the results shows, the improved approach can significantly reduce the average inquiries, thus achieves good real-time character.

We also measure the average attempts of each matching, defined as:

$$\text{average attempts} = \frac{\text{total inquiries of all the successful matches}}{\text{number of successful matches}}$$

Average attempts -1 indicates how many unnecessary inquiries is attempted before a success inquiry.

Table 3 shows result:

Trace	Baseline approach	Bloom Filter based approach
TPC-C	10.14	1.16
OLTP	5.26	1.03
kernel	11.78	1.07

Table 3: Average attempts

We can see that the bloom filter based approach nearly takes only one attempting inquiry for each success matching. It is indeed of high efficiency in avoiding making false attempts.

4.4 Disk Response Time

For horizontal comparison of the algorithm performance, we implement the readahead algorithm in our simulation environment. We feed the traces to the following three experiment environments respectively: 1) the one using association rule based prefetcher; 2) the one use readahead prefetching algorithm and 3) none prefetcher environment.

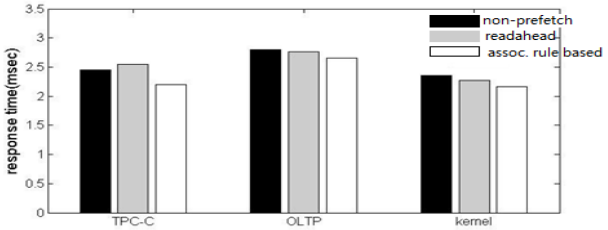


Figure 5: disk response time over 3 prefetching settings

Figure 5 shows response times produced by the three traces. In each bar group, the left bar is simulated with none prefetcher, middle bar is with sequential prefetcher, and right one is with our association rule guided prefetcher.

Our prefetcher can effectively decrease the response time over these three traces, especially on TPC-C and OLTP, considering the measurements decrease by 8%-10% comparing with the experiments with none prefetching. We

observe that when simulating using sequential policy on TPC-C trace, the response time even goes higher than the one with none prefetcher. These prefetched blocks are then recorded and analyzed, and we find it is the short sequential fragments in TPC-C trace triggers sequential prefetching too early and open many over-sized prefetching windows. Consequentlytoo many un-hit blocks are brought in, whichcauses cache thrashing. Although its performance can be promoted by tuning the policy, the response time cannot decrease as much as our approach achieves.

5 Conclusion

In this paper, we have proposed an association rule mining based prefetch scheme for disk storage systems.We have discussed the idea of using the correlation information of I/O access events in the block-level to direct the prefetcher operating for complex workloads. Therefore, an algorithm for extracting association rule is proposed to contribute to prefetching. To utilize the knowledge of the mined rules while response the I/O stream in real-time, an effective rule matching algorithm has been devised. The evaluation experiments on real workloads have shownthat our approach can efficiently discover a set of accurate prediction rules at an acceptable cost and achieve a good real-time feature. As a consequence, the performance of the storage system can be promoted by the association rule based prefetching.

References

[1] Patterson D A. Computer architecture: a quantitative approach[M]. Elsevier, 2011.
[2] Bryant R, David Richard O H, David Richard O H. Computer systems: a programmer's perspective[M]. Upper Saddle River: Prentice Hall, 2003.
[3] Agrawal N, Prabhakaran V, Wobber T, et al. Design Tradeoffs for SSD Performance[C]//USENIX Annual Technical Conference. 2008: 57-70.
[4] Strecker W D. Cache memories for PDP-11 family computers[J]. ACM SIGARCH Computer Architecture News, 1976, 4(4): 155-158.
[5] Chen Y, Byna S, Sun X H, et al. Hiding I/O latency with pre-execution prefetching for parallel applications[C]//High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for. IEEE, 2008: 1-10.
[6] Corbett P, Feitelson D, Hsu Y, et al. MPI-IO: A Parallel File I/O Interface for MPI Version 0.3[J]. 1995.
[7] Thakur R, Gropp W, Lusk E. On implementing MPI-IO portably and with high performance[C]//Proceedings of the sixth workshop on I/O in parallel and distributed systems. ACM, 1999: 23-32.
[8] Patterson R H, Gibson G A, Ginting E, et al. Informed prefetching and caching[M]. ACM, 1995.
[9] Yang C K, Mitra T, Chiueh T. A Decoupled Architecture for Application-Specific File Prefetching[C]//USENIX Annual Technical Conference, FREENIX Track. 2002: 157-170.
[10] Oly J, Reed D A. Markov model prediction of I/O requests for scientific applications[C]//Proceedings of the 16th international conference on Supercomputing. ACM, 2002: 147-155.
[11] Deshpande M, Karypis G. Selective Markov models for predicting Web page accesses[J]. ACM Transactions on Internet Technology (TOIT), 2004, 4(2): 163-184.
[12] Madhyastha T M, Reed D A. Input/output access pattern classification using hidden Markov models[C]//Proceedings of the fifth workshop on I/O in parallel and distributed systems. ACM, 1997: 57-67.
[13] Tran N, Reed D A. Automatic ARIMA time series modeling for adaptive I/O prefetching[J]. IEEE Transactions on parallel and distributed systems, 2004, 15(4): 362-377.
[14] Dangelmayr G, Gadaleta S, Hundley D, et al. Time series prediction by estimating Markov probabilities through topology preserving maps[C]//SPIE's International Symposium on Optical Science, Engineering, and Instrumentation. International Society for Optics and Photonics, 1999: 86-93.
[15] Rabiner L, Juang B. An introduction to hidden Markov models[J]. ieee assp magazine, 1986, 3(1): 4-16.
[16] Fengguang W U, Hongsheng X I, Chenfeng X U. On the design of a new linux readahead framework[J]. ACM SIGOPS Operating Systems Review, 2008, 42(5): 75-84.
[17] Wu F, Xi H, Li J, et al. Linux readahead: less tricks for more[C]//Proceedings of the Linux Symposium. 2007, 2: 273-284.
[18] Byna S, Chen Y, Sun X H, et al. Parallel I/O prefetching using MPI file caching and I/O signatures[C]//Proceedings of the 2008 ACM/IEEE conference on Supercomputing. IEEE Press, 2008: 44.
[19] Gu P, Wang J, Zhu Y, et al. A novel weighted-graph-based grouping algorithm for metadata prefetching[J]. IEEE Transactions on Computers, 2010, 59(1): 1-15.
[20] Curewitz K M, Krishnan P, Vitter J S. Practical prefetching via data compression[C]//ACM SIGMOD Record. ACM, 1993, 22(2): 257-266.
[21] Griffioen J, Appleton R. Reducing File System Latency using a Predictive Approach[C]//USENIX summer. 1994: 197-207.
[22] Kroeger T M, Long D D E. Design and Implementation of a Predictive File Prefetching Algorithm[C]//USENIX Annual Technical Conference, General Track. 2001: 105-118.