Mining Block Correlations to Improve Storage Performance

ZHENMIN LI, ZHIFENG CHEN and YUANYUAN ZHOU
University of Illinois at Urbana-Champaign

Block correlations are common semantic patterns in storage systems. They can be exploited for improving the effectiveness of storage caching, prefetching, data layout, and disk scheduling. Unfortunately, information about block correlations is unavailable at the storage system level. Previous approaches for discovering file correlations in file systems do not scale well enough for discovering block correlations in storage systems.

In this article, we propose two algorithms, *C-Miner* and *C-Miner**, that use a data mining technique called *frequent sequence mining* to discover block correlations in storage systems. Both algorithms run reasonably fast with feasible space requirement, indicating that they are practical for dynamically inferring correlations in a storage system. *C-Miner* is a direct application of a frequent-sequence mining algorithm with a few modifications; compared with *C-Miner*, *C-Miner** is redesigned for mining block correlations by making concessions for the specific problem of long sequences in storage system traces. Therefore, *C-Miner** can discover 7–109% more correlation rules within 2–15 times shorter time than *C-Miner*. Moreover, we have also evaluated the benefits of block correlation-directed prefetching and data layout through experiments. Our results using real system workloads show that correlation-directed prefetching and data layout can reduce average I/O response time by 12–30% compared to the base case, and 7–25% compared to the commonly used sequential prefetching scheme for most workloads.

Categories and Subject Descriptors: D.4.2 [**Operating Systems**]: Storage Management; C.4 [**Performance of Systems**]; H.2.8 [**Database Management**]: Database Applications—*Data mining*; H.3.m [**Information Storage and Retrieval**]: Miscellaneous

General Terms: Algorithms, Management, Performance

Additional Key Words and Phrases: Storage management, file system management, mining methods and algorithms, block correlations

This research is supported by the NSF CCR-0305854 Grant and IBM CAS Fellowship. Our experiments were conducted on equipment provided through the IBM SUR Grant.

An earlier version of this article appeared in *Proceedings of the 3rd USENIX Conference on File and Storage Technologies, 2004* (FAST '04, San Francisco, CA). We have added significant extensions in this article. In particular, besides *C-Miner*, we also present a new algorithm, called *C-Miner**, that is more suitable for discovering block correlations from storage system traces. In addition, we also present the results of *C-Miner**, which we compare with those of *C-Miner*. Furthermore, besides OLTP and an old file system workload, we also studied other types of workloads including a more recent file system workload (Cello-99) and a DSS workload (TPC-H).

Authors' address: Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801; email: {zli4,zchen9,yyzhou}@uiuc.edu.

1. INTRODUCTION

To satisfy the growing demand for storage, modern storage systems are becoming increasingly intelligent. For example, the IBM Storage Tank system [IBM 2002] consists of a cluster of storage nodes connected using a storage area network. Each storage node includes processors, memory, and disk arrays. An EMC Symmetric server contains up to *eighty* 333-MHz microprocessors with up to 4–64 GB of memory as the storage cache [EMC Corporation 1999]. Figure 1 gives an example architecture of modern storage systems. Many storage systems also provide virtualization capabilities to hide disk layout and configurations from storage clients [Lee and Thekkath 1996; Anthes 2002].

Unfortunately, it is not an easy task to exploit the increasing intelligence in storage systems. One primary reason is the narrow I/O interface between storage applications and storage systems. In such a simple interface, storage applications perform only block read or write operations without any indication of access patterns or data semantics. As a result, storage systems can only manage data at the block level without knowing any semantic information such as the semantic correlations between blocks. Therefore, much previous work had to rely on simple patterns such as temporal locality, sequentiality, and loop references to improve storage system performance, without fully exploiting its intelligence. This motivates a more powerful analysis tool to discover more complex patterns, especially semantic patterns, in storage systems.

Block correlations are common semantic patterns in storage systems. Many blocks are correlated by semantics. For example, in a database that uses index trees such as B-trees to speed up query performance, a tree node is correlated to its parent node and its ancestor nodes. Similarly, in a file server-backend storage system, a file block is correlated to its inode block. Correlated blocks tend to be accessed relatively close to each other in an access stream.

Exploring block correlations is very useful for improving the effectiveness of storage caching, prefetching, data layout, and disk scheduling. For example, at each access, a storage system can prefetch correlated blocks into its storage cache so that subsequent accesses to these blocks do not need to access disks, which is several orders of magnitude slower than accessing directly from a storage cache. As self-managing systems are becoming ever so important, capturing block correlations would enhance the storage system's knowledge about its workloads, a necessary step toward providing self-tuning capability. Furthermore, compared with capturing semantic patterns at the file level, exploring block correlations is more general since it works for all types of storage clients without any a priori knowledge, including both file systems and databases. Additionally, exploring correlations at block level can be integrated with I/O scheduling more effectively than at file level.

Unfortunately, information about block correlations are unavailable at a storage system because a storage system exports only block interfaces. Since databases or file systems are typically provided by vendors different from those of storage systems, it is quite difficult and complex to extend the block I/O interface to allow upper levels to inform a storage system about block correlations. Recently, Arpaci-Dusseau et al. proposed a very interesting approach

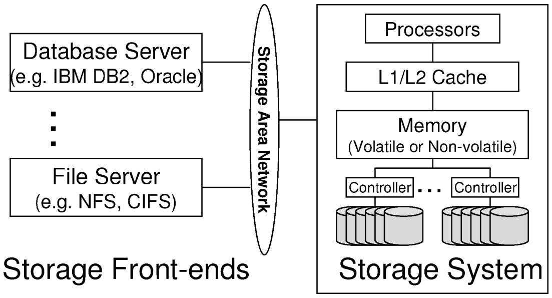


Fig. 1. Example of modern storage architecture.

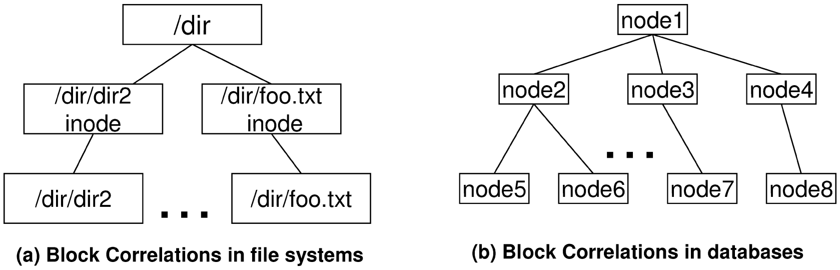
called *semantically-smart disk systems* (SDS) [Sivathanu et al. 2003] by using a "gray-box" technology to infer data structure and categorize data in storage systems. However, this approach requires probing in the front-end and assumes that the front-ends conform to the FFS-like file system layout.

An alternative approach is to infer block correlations fully transparently inside a storage system by only observing access sequences. This approach does not require any probing from a front-end and also makes no assumption about the type of the front-ends. Therefore, this approach is more general and can be applied to storage systems with any front-end file systems or database servers. Semantic distances [Kuenning 1994; Kuenning and Popek 1997] and probability graphs [Griffioen and Appleton 1994, 1995] are such "black-box" approaches. They are quite useful in discovering file correlations in file systems (see Section 2.3 for more details).

This article proposes *C-Miner*, a method which applies a data mining technique called *frequent sequence mining* to discover block correlations in storage systems. Specifically, we have modified a recently proposed data mining algorithm called *CloSpan* [Yan et al. 2003] to find block correlations in several storage traces collected in real systems. In order to improve performance and accuracy, we further develop a more suitable and tailored algorithm, called *C-Miner**, for discovering correlations from *a long sequence* such as the storage system traces. To the best of our knowledge, our technique is the first approach to infer block correlations involving multiple blocks. Furthermore, both algorithms are more scalable and space-efficient than previous approaches. They run reasonably fast with reasonable space overhead, indicating that they are practical for dynamically inferring correlations in a storage system.

Moreover, we have also evaluated the benefits of block correlation-directed prefetching and disk data layout using the real system workloads. Compared to the base case, this scheme reduces the average response time by 12% to 30%; compared to the sequential prefetching scheme, it also reduces the average response time by 7% to 25% for most workloads except TPC-H.

The article is organized as follows. In the next section, we briefly describe block correlations, the benefits of exploiting block correlations, and approaches to discover block correlations. In Section 3, we present our data mining method to discover block correlations including *C-Miner* and *C-Miner**. Section 4 discusses how to take advantage of block correlations in two specific



(a) Block Correlations in file systems

(b) Block Correlations in databases

Fig. 2. Examples of block correlations.

applications, prefetching and disk layout. Section 5 presents our experimental results. Section 7 discusses the related work and Section 8 concludes the article.

2. BLOCK CORRELATIONS

2.1 What are Block Correlations?

Block correlations commonly exist in storage systems. Two or more blocks are correlated if they are "linked" together semantically. For example, Figure 2(a) shows some block correlations in a storage system which manages data for a file system server. In this example, a directory block "/dir" is directly correlated to the inode block of "/dir/foo.txt," which is also directly correlated to the file block of "/dir/foo.txt." Besides direct correlations, blocks can also be correlated indirectly through another block. For example, the directory block "/dir" is indirectly correlated to the file block of "/dir/foo.txt." Figure 2(b) shows block correlations in a database-backend storage system. Databases commonly use a tree structure such as B-tree or B*-tree to store data, in which a node is directly correlated to its parent and children and also indirectly correlated to its ancestor and descendant nodes.

Unlike other access patterns such as temporal locality, block correlations are inherent in the data managed by a storage system. Access patterns such as temporal locality or sequentiality depend on workloads and can therefore change dynamically, whereas block correlations are relatively more stable and do not depend on workloads, but rather on data semantics. When block semantics are changed (for example, a block is reallocated to store other data), some block correlations may be affected. In general, block semantics are more stable than workloads, especially in systems that do not have very bursty deletion and insertion operations that can significantly change block semantics. As we will show in Section 5.5, block correlations can remain stable for several days in file systems.

Correlated blocks are usually accessed very close to each other. This is because most storage front-ends (database servers or file servers) usually follow semantic "links" to access blocks. For example, a file system server needs to access an inode block before accessing a file block. Similarly, a database server first needs to access the parent before accessing its children. Due to the interleaving of requests and transactions, these I/O requests may not be always consecutive in the access stream received by a storage system, but they should be relatively close within a short distance from each other.

Spatial locality is a simple case of block correlations. An access stream exhibits spatial locality if, after a block is accessed, other blocks that are near it are likely to be accessed in the near future. This is based on the observation that a block is usually semantically correlated to its neighboring blocks. For example, if a file's blocks are allocated in disks consecutively, these blocks are correlated to each other. Therefore, in some workloads, these blocks are likely accessed one after another.

However, many correlations are more complex than spatial locality. For example, for a file system server, an inode block is not necessarily allocated contiguous with its file blocks, and a directory block is usually allocated separately from the inode blocks of the files in this directory on disks [Sivathanu et al. 2003]. Therefore, although accesses to these correlated blocks are close to each other in the access stream, they do not exhibit good spatial locality because these blocks are far away from each other in the disk layout and even on different disks.

In some cases, a block correlation may involve more than two blocks. For example, a three-block correlation might be: if *both* a and b are accessed recently, c is very likely to be accessed in a short period of time. Basically, a and b are correlated to c, but a or b alone may not be correlated to c. To give a real instance of this multiblock correlation, let us consider a B* tree which also links all the leaf nodes together. a, b and c are all leaf nodes. If a is accessed, the system cannot predict that c is going to be accessed soon. However, if *a and b* are accessed one after another, it is likely that c will be accessed soon because it is likely that the front-end is doing a sequence scan of all the leaf nodes, which is very common in decision-support system (DSS) workloads [Barroso et al. 1998; Zhang et al. 2003].

2.2 Exploiting Block Correlations

Block correlations can be exploited to improve storage system performance. First, correlations can be used to direct prefetching. For example, if a strong correlation exists between blocks, these two blocks can be fetched together from disks whenever one of them is accessed. The disk read-ahead optimization is an example of exploiting the simple sequential block correlations by prefetching subsequent disk blocks ahead of time. Several studies [Smith 1978a; Choi et al. 2000; Kim et al. 2000] have shown that using even these simple sequential correlations can significantly improve the storage system performance. Our results in Section 5.4 demonstrate that prefetching based on block correlations can improve the performance much better than such simple sequential prefetching in most cases.

A storage system can also lay out data in disks according to block correlations. For example, a block can be collocated with its correlated blocks so that they can be fetched together using just one disk access. This optimization can reduce the number of disk seeks and rotations, which dominate the average disk access latency. With correlation-directed disk layouts, the system only needs to pay a one-time seek and rotational delay to get multiple blocks that are likely to be accessed soon. Previous studies [Schindler et al. 2002; Sivathanu et al. 2003] have shown promising results in allocating correlated file blocks on the same track to avoid track-switching costs.

Correlations can also be used to direct storage caching. For example, a storage cache can "promote" or "demote" a block after its correlated block is accessed or evicted. After an access to block A, blocks that are correlated to A are likely to be accessed very soon. Therefore, a cache replacement algorithm can specially "mark" these blocks to avoid being evicted. Similarly, after a block A is evicted, blocks that are correlated to A are not very likely to be accessed soon so it might be OK to also evict these blocks in subsequent replacement decisions. The storage cache can also give higher priority to those blocks that are correlated to many other blocks. Therefore, for databases that use tree structures, it would achieve a similar effect as the DBMIN cache replacement algorithm that is specially designed for database workloads [Chou and DeWitt 1993]. This algorithm gives higher priority to root blocks or high-level index blocks to stay in a database buffer cache.

Besides performance, block correlations can also be used to improve storage system security, reliability, and energy-efficiency. For example, malicious clients access the storage system in a very different pattern from the normal clients. By catching abnormal block correlations in an access stream, the storage system can detect such kind of malicious users. When a file block is archived to a tertiary storage, its correlated blocks may also need to be backed up in order to provide consistency. In addition, storage power management schemes can also take advantage of block correlations by clustering correlated blocks in the same disk so it is possible for other disks to transition into standby mode [Carrera et al. 2003].

The experiments in this study focused on demonstrating the benefits of exploiting block correlations in improving storage system performance. The usages for security, reliability, and energy-efficiency remain as our future work.

2.3 Obtaining Block Correlations

There can be three possible approaches to obtain block correlations in storage systems. These approaches trade transparency and generality for accuracy at different degrees. The "black box" approach is most transparent and general because it infers block correlations without any assumption or modification to storage front-ends. The "gray box" approach does not need modifications to front-end software but makes certain assumptions about front-ends and also requires probing from front-ends. The "white box" approach completely relies on front-ends to provide information and therefore has the most accurate information but is least transparent.

"Black Box" approaches infer block correlations completely inside a storage system, without any assumption on the storage front-ends. One commonly used method of this approach is to infer block correlations based on accesses. The observation is that correlated blocks are usually accessed relatively close to each other. Therefore, if two blocks are almost always accessed together within a short access distance, it is very likely that these two blocks are correlated to each other. In other words, it is possible to automatically infer block correlations in a storage system by dynamically analyzing the access stream.

In the field of networked or mobile file systems, researchers have proposed semantic distance (SD) [Kuenning 1994; Kuenning and Popek 1997] or probability graphs [Griffioen and Appleton 1994, 1995] to capture file correlations in file systems. The main idea is to use a graph to record the number of times two items are accessed within a specified access distance. In an SD graph, a node represents an accessed item B_1 with edges linking to other items. The weight of each edge (B_1, B_2) is the number of times that B_2 is accessed within the specified lookahead window of B_1's access. So if the weight for an edge is large, the corresponding items are probably correlated.

The algorithm to build the SD graph from an access stream works like this: suppose the specified lookahead window size is 100, that is, accesses that are less than 100 accesses apart are considered to be “close” accesses. Initially the probability graph is empty. The algorithm processes each access one after another. The algorithm always keeps track of the items of most recent 100 accesses in the current sliding window. When an item B is accessed, it adds node B into the graph if it is not in the graph yet. It also increments the weight of the edge (B_i, B) for any B_i accessed during the current window. If such an edge is not in the graph, it adds this edge and sets the initial weight to be 1. After the entire access stream is processed, the algorithm rescans the SD graph and only records those correlations with weights larger than a given threshold.

Even though probability graphs or SD graphs work well for inferring file correlations in a file system, they, unfortunately, are not practical for inferring block correlations in storage systems because of two reasons. (1) *Scalability problem*: a semantic distance graph requires one node to represent each accessed item and also one edge to capture each non-zero-weight correlation. When the system has a huge number of items as in a storage system, an SD graph is too big to be practical. For instance, if we assume the specified window size is 100, it may require more than 100 edges associated with each node. Therefore, one node would occupy at least $100 \times 8 = 800$ (assuming each edge requires 8 bytes to store the weight and the disk block number of B_2). For a small storage system with only 80 GB and a block size of 8 KB, the probability graph would occupy 8 GB, 10% of the storage capacity. Besides space overheads, building and searching such a large graph would also take a significantly large amount of time. (2) *Multiblock correlation problem*: these graphs cannot represent correlations that involve more than two blocks. For example, the block correlations described at the end of the Section 2.1 cannot be conveniently represented in a semantic distance graph. Therefore, these techniques can lose some important block correlations.

In this article, we present a practical black box approach that uses a data mining method to automatically infer both dual-block and multiblock correlations in storage systems. In Section 3, we describe our approach in detail.

“Gray Box” approaches were investigated by Arpaci-Dusseau et al. in [2001] They developed three gray-box information and control layers between a client and the OS, and combined algorithmic knowledge, observations, and inferences to collect information.

The gray-box idea has been explored by Sivathanu et al. [2003] in storage systems to automatically obtain file-system knowledge. The main idea is to probe from a storage front-end by performing some standard operations and then observing the triggered I/O accesses to the storage system. It works very well for file systems that conform to FFS-like structure (if the front-end security is not a concern). The advantage of this approach is that it does not require any modification to the storage front-end software. The tradeoff is that it requires the front-end to conform to specific disk layouts such as FFS-like structure.

"White Box" approaches rely on storage front-ends to directly pass semantic information to obtain block correlations in a storage system. For example, the storage I/O interface can be modified using a higher-level, object-like interface [Gibson et al. 1998] so that correlations can be easily expressed using the object interface. The advantage with this approach is that it can obtain very accurate information about block correlations from storage front-ends. However, it requires modifying storage front-end software, some of which, such as database servers, are too large to be easily ported to object-based storage interface.

3. MINING FOR BLOCK CORRELATIONS

Data mining, also known as *knowledge discovery in databases* (KDD), has developed quickly in recent years due to the wide availability of voluminous data and the imminent need for extracting useful information and knowledge from them. Traditional methods of data analysis dependent on human handling cannot scale well to huge sizes of data sets. In this section, we first introduce some fundamental data mining concepts and analysis methods used in our article and then describe *C-Miner* and *C-Miner**, our algorithms for inferring block correlations in storage systems.

3.1 Frequent Sequence Mining

Different patterns can be discovered by different data mining techniques, including association analysis, classification and prediction, cluster analysis, outlier analysis, and evolution analysis [Han and Kamber 2001]. Among these techniques, association analysis can help discover correlations between two or more sets of events or attributes. Suppose there exists a strong association between events x and y, it means that if event x happens, event y is also very likely to happen. We use the association rule $x \rightarrow y$ to describe such a correlation between these two events.

Frequent sequence mining is one type of association analysis to discover frequent subsequences in a sequence database [Agrawal and Srikant 1995]. A subsequence is considered *frequent* when it occurs in at least a specified number of sequences (called *min_sup*) in the sequence database. A subsequence is not necessarily contiguous in an original sequence. For example, a sequence database D has five sequences:

$$D = \{abced, abcef, agbch, abijc, aklc\}.$$

The number of occurrences of subsequence *abc* is 4. We denote the number of occurrences of a subsequence as its *support*. Obviously, the smaller *min_sup* is, the more frequent subsequences the database contains. In the above example, if *min_sup* is specified as 5, only the subsequence *ac* is frequent; if *min_sup* is specified as 4, the frequent subsequences are

$$\{ab\colon 4, ac\colon 5, bc\colon 4, abc\colon 4\},$$

where the numbers are the supports of the subsequences.

Frequent sequence mining is an active research topic in data mining [Zaki 2001; Pei et al. 2001; Ayres et al. 2002] with broad applications, such as mining motifs in DNA sequences, analysis of customer shopping sequences, etc. To the best of our knowledge, our study is the first one that uses frequent sequence mining to discover patterns in storage systems.

Our algorithms are based on a recently proposed frequent sequence mining algorithm called *CloSpan* (Closed Sequential Pattern mining)[Yan et al. 2003]. The main idea of CloSpan is to find only closed frequent subsequences. A *closed sequence* is a subsequence whose support is different from that of its supersequences. In the above example, subsequence *ac* is closed because its support is 5, and the support of any one of its supersequences (for example, *abc* and *agc*, etc.) is no more than 4; on the other hand, subsequence *ab* is not closed because its support is the same as that of one of its supersequences, *abc*.

CloSpan only produces the closed frequent subsequences rather than all frequent subsequences since any nonclosed subsequences can be indicated by their supersequences with the same support. In the above example, the frequent subsequences are

$$\{a\colon 5, b\colon 4, c\colon 5, ab\colon 4, ac\colon 5, bc\colon 4, abc\colon 4\}$$

but we only need to produce the closed subsequences $\{ac\colon 5, abc\colon 4\}$. This feature significantly reduces the number of patterns generated, especially for long frequent subsequences. More details can be found in Pei et al. [2001] and Yan et al. [2003].

3.2 *C-Miner* : Our Basic Mining Algorithm

Frequent sequence mining is a good candidate for inferring block correlations in storage systems. One can map a block to an item, and an access sequence to a sequence in the sequence database. Using frequent sequence mining, we can obtain all the frequent subsequences in an access stream. A frequent subsequence indicates that the involved blocks are frequently accessed together. In other words, frequent subsequences are good indications of block correlations in a storage system.

One limitation with the original mining algorithm is that it does not consider the gap of a frequent subsequence. If a frequent sequence contains two accesses that are very far from each other in terms of access time, such a correlation is not interesting for our application. From the system's point of view, it is much more interesting to consider frequent access subsequences that are not far apart. For example, if a frequent subsequence *xy* always appears in the original sequence with a distance of more than 1000 accesses, it is not a very interesting pattern because it is hard for storage systems to exploit it. Further, such correlations are generally less accurate.

To address this issue, *C-Miner* restrict access distances. In order to describe how far apart two accesses are in the access stream, the access distance between

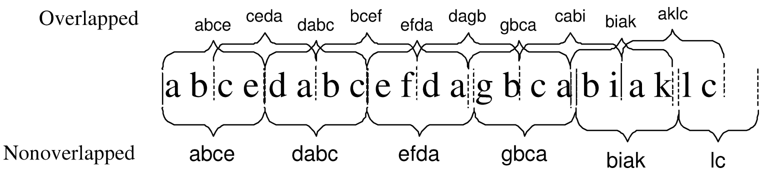


Fig. 3. Preprocessing traces using overlapped and nonoverlapped windows (cutting window size = 4).

them is denoted as *gap*, measured by the number of accesses between these two accesses. We specify a maximum distance threshold, denoted as *max_gap*. All the uninteresting frequent sequences whose gaps are larger than the threshold are filtered out. This is similar to the lookahead window used in the semantic distance algorithms.

3.2.1 *Preprocessing.* Existing frequent sequence mining algorithms including CloSpan are designed to discover patterns for a sequence database rather than a single long sequence of time-series information as in storage systems. To overcome this limitation, *C-Miner* preprocesses the access sequence (that is, the history access trace) by breaking it into fixed-size short sequences. The size of each short sequence is called *cutting window size*.

There are two ways to cut the long access stream into short sequences—overlapped cutting and nonoverlapped cutting. The overlapped cutting divides an entire access stream into many short sequences and leaves some overlapped regions between any two consecutive sequences. Nonoverlapped cutting is straightforward; it simply splits the access stream into access sequences of equal size.

Figure 3 illustrates how to cut the access stream *abcedabcefdagbcabiaklc* into short sequences with length of 4 using these two methods. Overlapped cutting may increase the number of occurrences for some subsequences if it falls in the overlapped region. In the example shown in Figure 3, using overlapped cutting results in 10 short sequences. The subsequences *bc* in *dabc* and *bcef* occurs only once in the original access stream, but now is counted twice since the short sequences, *dabc* and *bcef*, overlap with each other. It is quite difficult to determine how many redundant occurrences there are due to overlapping. Another drawback is that the overlapped cutting generates more sequences than nonoverlapped cutting. Therefore it takes the mining algorithm a longer time to infer frequent subsequences.

Using nonoverlapped cutting can, however, lead to loss of frequent subsequences that are split into two or more sequences, and therefore can decrease the support values of some frequent subsequences because some of their occurrences are split into two sequences. In the example shown in Figure 3, the nonoverlapped cutting results in only six sequences. The support for *ab* is 2, but the actual support in the original long sequence is 4. The lost support is because the last two occurrences are broken across cutting windows and are therefore not counted.

But the amount of lost information in the non-overlapped cutting scheme is usually quite small, especially if the cutting window size is relatively large.

```
**Algorithm:** MINING($s, D_s, min\_sup, L$)
**Input:** A frequent subsequence $s$, a set of subsequence $D_s$,
        support threshold $min\_sup$.
**Output:** The frequent sequence set $L$.
  insert $s$ to $L$.
  scan $D_s$ to find every frequent item $\alpha$ such that $s \diamond \alpha$ is frequent sequence,
    $D_{s \diamond \alpha} \leftarrow \{$ all maximum suffixes that can be concatenated with $s \diamond \alpha\}$.
  for each $\alpha$ do
      MINING($s \diamond \alpha, D_{s \diamond \alpha}, min\_sup, L$).

            *(Note: $s \diamond \alpha$ means to concatenate $s$ with $\alpha$.)*
```

Fig. 4. Mining algorithm.

Since *C-Miner* restricts the access distance of a frequent subsequence, only a few frequent subsequences may be split across multiple windows. Suppose the instances of a frequent subsequence are distributed uniformly in the access stream, the cutting windows size is w and the maximum access distance for frequent sequences is max_gap ($max\_gap \ll w$). Then, the probability that an instance of a frequent subsequence is split across two sequences is max_gap/w. For example, if the access distance is limited within 50, and the cutting window size is 500 accesses, the support value is lost by about 10%. Therefore, most frequent subsequences would still be considered frequent after nonoverlapped cutting. Based on this analysis, we used nonoverlapped cutting in our experiments.

3.2.2 *Core Algorithm.* *C-Miner* mines the sequence database and produces frequent subsequences, which can then be used to derive block correlations. *C-Miner* mainly consists of two stages: (1) generating a candidate set of frequent subsequences that includes all the closed frequent subsequences; and (2) pruning the nonclosed subsequences from the candidate set.

In the first stage, *C-Miner* generates a candidate set of frequent sequences using a depth-first search procedure. The pseudocode in Figure 4 shows the mining algorithm. In the algorithm, D_s is a suffix database which contains all the maximum suffixes of the sequences that contain the frequent subsequence s. For example, in the previous sequence database D, the suffix database of frequent subsequences ab is $D_{ab} = \{ced, cef, ch, ijc\}$.

There are two main ideas in *C-Miner* to improve the mining efficiency. The first one is based on an obvious observation that, if a sequence is frequent, all of its subsequences are frequent. For example, if abc is frequent, all of its subsequences $\{a, b, c, ab, ac, bc\}$ are frequent. Based on this observation, *C-Miner* recursively produces a longer frequent subsequence by concatenating every frequent item to a shorter frequent subsequence that has already been obtained in the previous iterations.

To better explain this idea, let us consider an example. In order to get the set L_n of frequent subsequences with length n, we can join the set L_{n-1} of frequent subsequences with length $n-1$ and the set L_1 of frequent subsequences with length 1. For example, suppose we have already computed L_1 and L_2 as shown below. In order to compute L_3, we can first compute L'_3 by concatenating a subsequence from L_2 and an item from L_1:

$$\begin{aligned} L_1 &= \{a, b, c\}; \\ L_2 &= \{ab, ac, bc\}; \\ L'_3 &= L_2 \times L_1 = \{abc, abb, abc, aca, acb, acc, bca, bcb, bcc\}. \end{aligned}$$

For greater efficiency, *C-Miner* does not join the sequences in L_2 with all the items in L_1. Instead, each sequence in L_2 is concatenated with only the frequent items in its suffix database. In our example, for the frequent sequence ab in L_2, its suffix database is $D_{ab} = \{ced, cef, ch, ijc\}$, and only c is the frequent item, so ab is only concatenated with c and we get a longer sequence abc that belongs to L'_3.

The second idea is used for efficiently evaluating whether a concatenated subsequence is frequent or not. It tries to avoid searching through the whole database. Instead, it checks with certain suffixes. In the above example, for each sequence s in L'_3, *C-Miner* checks whether it is frequent or not by searching the suffix database D_s. If the number of its occurrences is greater than *min_sup*, s is added into L_3, which is the set of frequent subsequences of length 3. *C-Miner* continues computing L_4 from L_3, L_5 from L_4, and so on until no more subsequences can be added into the set of frequent subsequences.

In order to mine frequent sequences more efficiently, *C-Miner* uses a technique that can efficiently determine whether there are new closed patterns in search subspaces and stop checking those unpromising subspaces. The basic idea is based on the following observation about a closed sequence property. In the algorithm's step 2, among all the sequences in D_s, if an item a always occurs before another item b, *C-Miner* does not need to search any sequences with prefix $s \diamond b$. The reason is that $\forall \gamma, s \diamond b \diamond \gamma$ is not closed under this condition. Take the previous sequence database as an example. a always occurs before b, so any subsequence with prefix b is not closed because it is also a subsequence with prefix ab. Therefore, *C-Miner* does not need to search the frequent sequences with prefix b because all these frequent sequences are included in the frequent sequences with prefix ab (e.g., bc is included in abc with support 4). Without searching these unpromising branches, *C-Miner* can generate the candidate frequent sequences much more efficiently.

3.2.3 *Generating Association Rules.*

C-Miner produces frequent sequences that indicate block correlations, but it does not directly generate the association rules in the form of $x_1x_2 \rightarrow y$, which is much easier to use in storage systems.

In order to convert the frequent sequences into association rules, *C-Miner* breaks each sequence into several rules. In order to limit the number of rules, *C-Miner* constrains the length of a rule (the number of items on the left side of a rule). For example, a frequent sequence abc may be broken into the following set of rules with the same support of abc: $\{a \rightarrow b, a \rightarrow c, b \rightarrow c, ab \rightarrow c\}$.

Different closed frequent sequences can be broken into the same rules. For example, both *abc* and *abd* can be broken into the same rule $a \rightarrow b$, but they may have different support values. The support of a rule is the *maximum* support of all corresponding closed frequent sequences.

3.2.4 *Confidence of Rules.* For each association rule, we need to evaluate its accuracy. For example, in the above example, *a* occurs five times, but *ab* only occurs four times; this means that when *a* is accessed, *b* is also accessed in the near future (within *max_gap* distance) with probability 80%. We call this probability the *confidence* of the rule. When we use an association rule to predict future accesses, its confidence indicates the expected prediction accuracy. Predictions based on low-confidence rules are likely to be wrong and may not be able to improve system performance. Worse still, they may hurt the system performance due to overheads and side-effects. Because of this, we use confidence as a constraint to filter out the rules with low confidence.

The *support* metric is different from *confidence*. For example, suppose *x* and *y* are accessed only once in the entire access stream and their accesses are within the *max_gap* distance, the confidence of the association rule $x \rightarrow y$ is 100% whereas its support is only 1. This rule is not very interesting because it happens rarely. On the other hand, if a rule has high support but very low confidence (e.g., 5%), it may not be useful because it is too inaccurate to be used for prediction. Therefore, in practice, we usually specify a minimum support threshold *min_sup* and a minimum confidence threshold *min_conf* in order to filter low-quality association rules.

3.3 *C-Miner**: Mining from a Single Long Sequence

As we mentioned in Section 3.2, the existing frequent sequence mining algorithms have some limitations for mining block correlations in storage systems. The major limitation is that the training dataset must be a sequence database that consists of a number of sequences. This is because the mining algorithms were originally designed for analyzing customer's behavior from a large number of transactions, which are quite different from storage systems. Each transaction is considered as a sequence, and each sequence in the sequence database is relatively short (usually shorter than 100). However, in our problem space of extracting block correlations, the input is a long sequence of access trace instead of a sequent database. Therefore, the original frequent sequence mining algorithms cannot be directly applied to these traces.

To get around the above problem, *C-Miner* breaks the long trace into short sequences, which can result in loss of some patterns if the items of a pattern are separated across two different short sequences. In order to limit the loss, the long sequence should be broken with a relatively large cutting window size. On the other hand, a large cutting window size can result in a large time overhead due to the inherent limitation of the original frequent sequence mining algorithms. Therefore, the cutting window size should be chosen carefully in order to achieve reasonable accuracy with reasonable overhead.

To better address this problem, we develop a new mining algorithm, called *C-Miner**, that can discover the frequent subsequences from a single long

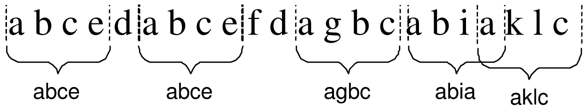


Fig. 5. Lookahead window for frequent item a (lookahead distance = 4).

sequence such as the storage system traces without cutting. The key difference from *C-Miner* is that *C-Miner** can count the supports of subsequences without breaking the long sequence, and therefore generate the candidate sets of frequent subsequences from a single long sequence.

The main idea of *C-Miner** is to count the support of a subsequence by scanning its corresponding *lookahead windows*. A lookahead window is a subsequence consisting of a specified number (*lookahead distance*) of items, and the subsequence begins with the frequent item that is under consideration. For example, Figure 5 shows the lookahead windows for the frequent item a in a single long sequence *abcedabcefdagbcabiaklc*. There are five lookahead windows for item a, each of which is composed of a four-item subsequence starting with a.

When *C-Miner** counts the support for a subsequence beginning with item α, it only scans all corresponding lookahead windows beginning with the same item α. For example, when *C-Miner** counts the support for subsequence ab, it scans the lookahead windows for frequent item a shown in Figure 5 and so the support is 4. Similarly, the support of a, ac, and abc can also be obtained by scanning these lookahead windows.

*C-Miner** has better accuracy and efficiency than both the nonoverlapped and overlapped cutting methods. Specifically, compared with nonoverlapped cutting as in *C-Miner*, *C-Miner** can obtain more block correlations from the trace. Since *C-Miner** does not use a fixed window to cut the single long subsequence into short sequences, no pattern information will be lost. In the above example, the last two supports of ab will be lost if using nonoverlapped cutting, but the lookahead windows will never separate the sequence ab across two windows if the gap between a and b is less than the lookahead distance. Since we only want to discover the frequent subsequences with the gap not larger than the threshold *max_gap*, *C-Miner** uses *max_gap* as the default lookahead distance, and therefore it does not lose any information we want to obtain. Furthermore, *C-Miner** is more efficient than *C-Miner* in terms of time overhead. In *C-Miner*, in order to limit the loss due to cutting, the cutting window size should be much larger than *max_gap* and it may lead a large time overhead. In contrast, the lookahead distance in *C-Miner** is equal to *max_gap* so that it can avoid this time overhead.

Compared with the overlapped cutting method, *C-Miner** can produce more accurate support values because each occurrence of subsequences is only counted as one support, whereas some subsequences may be counted for several times due to duplicates in the latter. Further more, *C-Miner** is faster than the overlapped cutting method because each occurrence of a subsequence needs to scan only once within a lookahead window, while the latter needs scan more than once due to redundancy.

3.4 Efficiency of *C-Miner* and *C-Miner**

Compared with other methods such as probability graphs or SD graphs, *C-Miner* and *C-Miner** can find more correlations, especially those multiblock correlations. From our experiments, we find that these multiblock correlations are very useful for systems.

For dual-block correlations, which can also be inferred using previous approaches, our algorithms, *C-Miner* and *C-Miner**, are more efficient. First, our algorithms are much more space efficient than SD graphs because they do not need to maintain the information for nonfrequent sequences, whereas SD graphs need to keep the information for every block during the graph building process. Second, in terms of time complexity, our algorithms are the same ($O(n)$) as SD. But in practice, since our algorithms have much smaller memory footprint size, it is more efficient and can run in a cheap uniprocessor machine with moderate memory size as used in our experiments.

Other frequent sequence mining algorithms such as PrefixSpan [Pei et al. 2001] can also find long frequent sequences. Compared with these frequent sequence mining algorithms, *C-Miner* and *C-Miner** are more efficient for discovering long frequent sequences because they not only avoid searching the nonfrequent sequences while generating longer sequences, but also prune all the unpromising searching branches according to the closed sequence property, as we have discussed. *C-Miner* and *C-Miner** can outperform PrefixSpan by an order of magnitude for some datasets.

4. APPLICATIONS

4.1 Correlation-Directed Prefetching (CDP)

As we mentioned in Section 2.2, the block correlation information inferred by *C-Miner* or *C-Miner** can be used to prefetch more intelligently. Assume that we have obtained a block correlation rule: if block b_1 is accessed, block b_2 will also be accessed soon within a short distance (of length *gap*) with a certain confidence (probability). Based on this rule, when there is an access to block b_1, we can prefetch block b_2 into the storage cache since it will probably be accessed soon.

Several design issues should be considered while using block correlations for prefetching. One of the most important issues is how to effectively share the limited size cache for both caching and prefetching. If prefetching is too aggressive, it can pollute the storage cache and may even degrade the cache hit ratio and system performance. This problem has been investigated thoroughly by previous work [Cao et al. 1994, 1995; Patterson et al. 1995]. We therefore do not investigate it further in our article. In our simulation experiments, we simply fixed the cache size for prefetched data so it did not compete with nonprefetching requests. However, the total cache size was fixed at the same value for the system with and without prefetching in order to have a fair comparison.

Another design issue is the extra disk load imposed by prefetch requests. If the disk load is too heavy, the disk utilization is close to 100%. In this case, prefetching can add significant overheads to demand requests, canceling out the benefits of improved storage cache hit ratio. Two methods can be used to alleviate this problem. The first method is to differentiate between demand requests and prefetch requests by using a priority-based disk scheduling scheme. In particular, the system uses two waiting queues in the disk scheduler: critical and noncritical. All the demand requests are issued to the critical queue, while the prefetch requests are issued to the noncritical queue which has lower priority.

The other method is to throttle the prefetch requests to a disk if the disk is heavily utilized. Since the correlation rules have different confidences, we can set a confidence threshold to limit the number of rules used for prefetching. All the rules with confidence lower than the threshold are ignored. Obviously, the higher the threshold is, the fewer the rules are used; therefore, CDP acts less aggressively. In order to adjust the threshold to make prefetching adapt to the current disk workload, we keep track of the current load on each disk. When the workload is too high, say the disk utilization is more than 80%, we increase the confidence threshold for correlation rules that direct the issuing of prefetch requests to this disk. Once the disk load drops down to a low level, say the utilization is less than 50%, we decrease the confidence threshold for correlation rules so that more rules can be used for prefetching. By doing this, the overhead on disk bandwidth caused by prefetches is kept within an acceptable range.

4.2 Correlation-Directed Disk Layout

Block correlations can also help lay out data on disks to improve performance as described in Section 2.2. We can lay out the blocks on disks based on block correlations like that: if we know a correlation *abcd* from *C-Miner* or *C-Miner**, we can try to allocate them contiguously in a disk. Whenever any one of these blocks is read, all four blocks are fetched together into the storage cache using one disk access. Since some blocks may appear in several patterns, we allocate the block based on the rules with highest support value.

One of the main design issues is how to maintain the directory information and reorganize data without an impact on the foreground workload. After reorganizing disk layouts, we need to map logical block numbers to new physical block numbers. The mapping table might become very large. Some previous work has studied these issues and shown that disk layout reorganization is feasible to implement [Salmon et al. 2003]. They proposed a two-tiered software architecture to combine multiple disk layout heuristics so that it adapts to different environments. Block correlation-directed disk layout can be one of the heuristics in their framework. Due to space limitation, we do not discuss this issue further.

5. SIMULATION RESULTS

5.1 Evaluation Methodology

To evaluate the benefits of exploiting block correlations in block prefetching and disk data layout, we use trace-driven simulations with several large disk traces collected in real systems. Our simulator combines the widely used DiskSim simulator [Ganger 1995] with a storage cache simulator, CacheSim, to simulate a complete storage system. Since most of the current storage systems use least recently used (LRU) replacement policy due to its simplicity, we used LRU in our experiments. Accesses to the simulated storage system first go through a storage cache and only read misses or writes access physical disks. The simulated disk specification is similar to that of the 10,000-rev/min IBM Ultrastar 36Z15. The parameters are taken from the disk's data sheet [Carrera et al. 2003].

Our experiments used the following six *real system* traces:

—*Cello-92 and Cello-96* were collected at Hewlett-Packard Laboratories in 1992 and 1996 [Ruemmlerler and Wilkes 1993a, 1993b]. They captured all low-level disk I/O performed by the system. We used the traces gathered on Cello, which is a timesharing system used by a group of researchers at HP Lab to do simulations, compilation, editing, and email. The traces include the accesses to 8 and 20 disks from multiple users and miscellaneous applications, respectively. They contain a lot of sequential access patterns, so the simple sequential prefetching approaches can significantly benefit from them.

—*Cello-99* is a more recent file system workload that was collected in 1999, and thereby it represents the modern workloads. It includes the accesses to 22 disks.

—*TPC-C Trace* is an I/O trace collected on a storage system connected to a Microsoft SQL Server via storage area network. The Microsoft Server SQL clients connect to the Microsoft SQL Server via Ethernet and run the TPC-C benchmark [Leutenegger and Dias 1993] for 2 h. The database consists of 256 warehouses and the footprint is 60 GB, and the storage system employs a RAID of four disks. More detailed descriptions of this trace can be found in Zhou et al. [2001] and Chen et al. [2003].

—*OLTP* is a trace of an OLTP application running at a large financial institution. It was made available by the Storage Performance Council [Storage Performance Council 2004]. The disk subsystem is composed of 19 disks.

—*TPC-H Trace* is another TPC benchmark trace that is collected on a storage system similar to TPC-C trace. The main difference is that it represents the DSS workload, and the sequential accesses is the dominant access patterns in TPC-H. The footprint of the trace is 12 GB, and the storage system employs a RAID of 20 disks.

All the traces were collected after filtering through a first-level buffer cache such as the database server cache. Fortunately, unlike other access patterns, such as temporal locality that can be filtered by large first-level buffer caches, most block correlations can still be discovered at the second level. Only those correlations involving "hot" blocks that always stay at the first level can be lost at the second level. However, these correlations are not useful to exploit anyway since "hot" blocks are kept at the first level and therefore are rarely accessed at the second level.

In our experiments, we used only the first half part of the trace to mine block correlations using *C-Miner* and *C-Miner**. Using these correlation rules, we evaluated the performance of correlation-directed prefetching and data layout

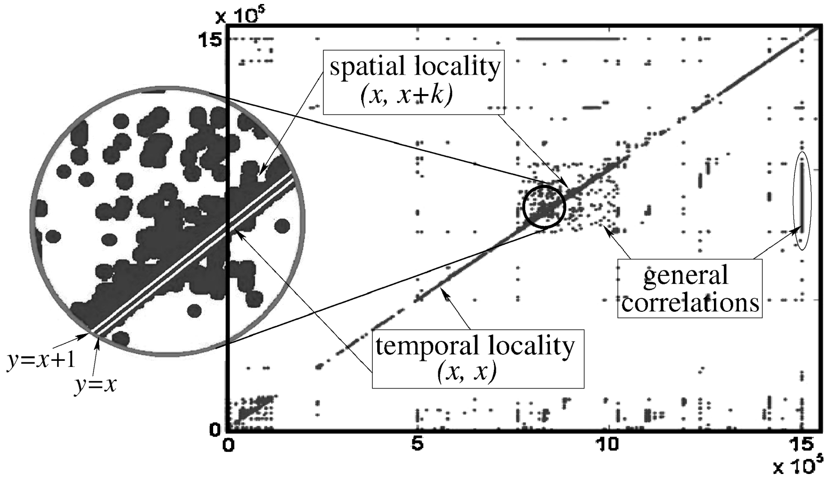


Fig. 6. Block correlations mined form the Cello-96 trace. If there is an association rule $x \rightarrow y$, we plot a corresponding point at (x, y). Therefore, each point (x, y) in the graph indicates a correlation between blocks x and y.

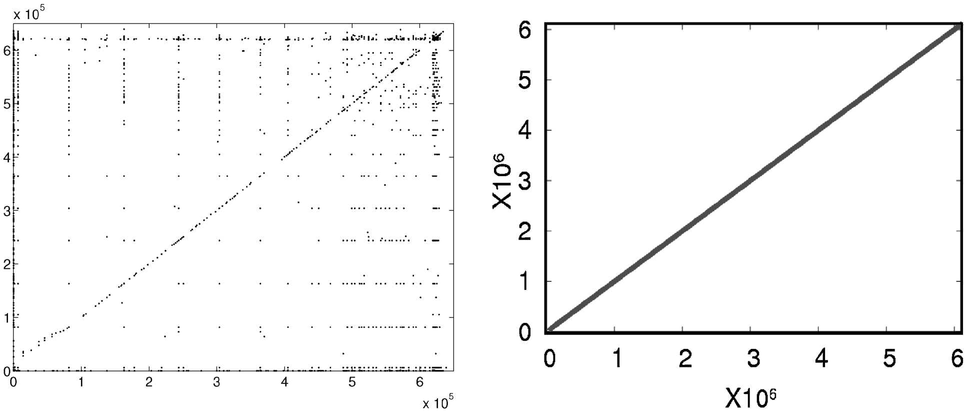
using the rest of the traces. The correlation rules were kept unchanged during the evaluation phase. For example, in Cello-92, we used the first 3 days' trace (1992.5.30–6.1) to mine block correlations and used the following 4 days to evaluate the correlation-directed prefetching and data layout. The reason for doing this was to show the stable characteristic of block correlations and predictive powers of our method.

To provide a more fair comparison, we also implemented the commonly used sequential prefetching scheme. At nonconsecutive misses to disks, the system also issues a prefetch request to load 16 consecutive blocks if the miss belongs to sequential accesses detected by a simple detector. We also tried prefetching more or fewer blocks, but the results were similar or worse.

5.2 Visualization of Block Correlations

5.2.1 *Correlations in Real System Traces.* Figures 6 and 7 plot the block correlations discovered by our technique from the Cello-96, TPC-C, and TPC-H traces. Since multiblock correlations are difficult to visualize, we plot only dualblock correlations. Each point (x, y) in the graphs indicates a correlation between blocks x and y. Since the traces contain multiple disks' accesses, we plot the disk block address using a unified continuous address space by plotting one disk address space after another.

Simple patterns such as temporal locality can be demonstrated in such a correlation graph. For example, temporal locality is indicated by the diagonal line as shown in Figure 6. This is because the temporal locality can be represented by an association rule $x \rightarrow x$, which means that if block x is accessed, this block will be accessed again soon. Hence, the points showing temporal locality are located on the diagonal line, $y = x$, in the correlation graphs. As shown on Figure 6, the Cello-96 trace has reasonable temporal locality.



(a) *TPC-C trace* (b) *TPC-H trace*

Fig. 7. Block correlations mined from the TPC-C and TPC-H traces.

Simple patterns such as spatial locality can also be demonstrated in such a correlation graph. It is indicated by dark areas around the diagonal line as shown in Figure 6. This is because the spatial locality can be represented by an association rule $x \rightarrow (x \pm k)$ where k is a small number, which means that if block x accessed, its neighbor blocks are likely to be accessed soon. Since k is small, the points $(x, x \pm k)$ are around the diagonal line, as shown on the Cello-96 traces in Figure 6. Figure 7(a) shows that the TPC-C trace does not have such an apparent characteristic, indicating TPC-C does not have strong spatial locality. In contrast, Figure 7(b) shows that all the patterns discovered in TPC-H are sequential accesses, which is one of the characteristics of the DSS workload.

Some more complex patterns can also be seen from correlation graphs. For example, in Figure 6, there are many horizontal or vertical lines, indicating some blocks are correlated to many other blocks. Because this is a database I/O trace, these hot blocks with many correlations are likely to be the root of trees or subtrees. In the next subsection, we visualize block correlations specifically for tree structures.

5.2.2 *Correlations in B-tree.* In order to demonstrate the capability of our technique to discover semantics in a tree structure, we used a synthetic trace that simulates a client that searches data in a B-tree data structure, which is commonly used in databases. The B-tree maintained the indices for 5000 data items, and each block has space for four search-key values and five pointers. We performed 1000 searches. To simulate a real-world situation where some "hot" data items are searched more frequently than others, searches were not uniformly distributed. Instead, we used a Zipf distribution and 80% of the searches were to 100 "hot" data items.

The block correlations mined from the B-tree trace are visualized in Figure 8. Note here constructing this tree does not take any semantic information from the application (the synthetic trace generator). The edges between nodes are reconstructed purely based on block correlations. Due to the space limitation,

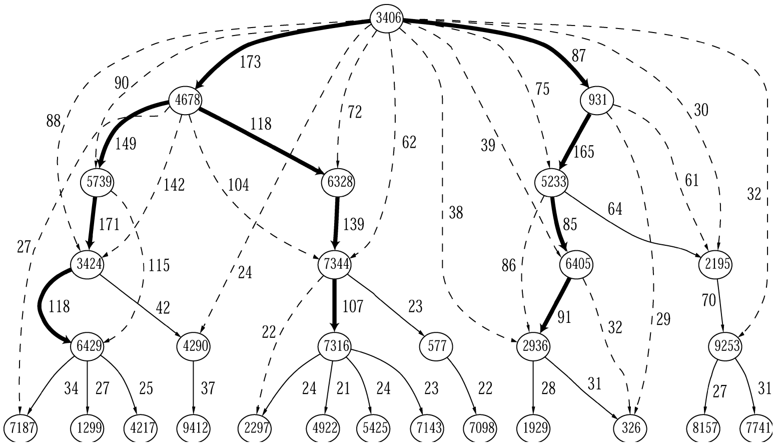


Fig. 8. Block correlations in B-tree. The number on an edge is the support value for the corresponding correlation. The dashed lines indicate the correlation between a node and its descendants other than its children. The highlighted lines are the correlations with *support* $\geq$ 80. Note that correlations with *support* $<$ 20 are not produced by *C-Miner* (*min_sup* $=$ 20) in order to make the tree reasonably small and sparse for plotting.

Table I. Correlation Rules and Mining Overheads (The rules are given with *confidence* $\geq$10%. The number following "# of rules" for *C-Miner** is the ratio between "# of rules" discovered by *C-Miner** and *C-Miner*.)

	C-Miner			*C-Miner**		
Training Trace	# of rules (10^3)	Time (s)	Space (MB)	# of rules (10^3)	Time (s)	Space (MB)
Cello-92 (3 days)	228	7800	3.1	335 (1.47×)	513	599
Cello-96 (1 day)	514	2089	4.6	608 (1.18×)	979	355
Cello-99 (1 day)	512	6060	8.5	1070 (2.09×)	2135	656
TPC-C (1 h)	235	3355	9.2	252 (1.07×)	1414	14
TPC-H (1 h)	24	49	4.4	306 (12.75×)	74	4.7
OLTP (2.5 h)	186	174	173	269 (1.45×)	40	455

we only show part of the correlations. Each rule $x \rightarrow y$ is denoted as a directed edge with support as its weight. The figure illustrates that the block correlations implicate a tree-like structure. Also note that our approach to obtaining block correlations is fully transparent without any assumption on storage front-ends.

5.3 Data Mining Overhead

Table I shows the running time and space overheads for mining different traces using *C-Miner* and *C-Miner** as described in Sections 3.2 and 3.3. Both algorithms were run on an Intel Xeon 2.4-GHz machine and Linux 2.4.20. The time and space overhead did not depend on the confidence of rules, as we discussed in Section 3.2, but the number of rules did.

The results show that *C-Miner* can effectively and practically discover block correlations for different workloads. For example, it takes less than 1 h to discover half a million association rules from the Cello-96 trace that contains a full-day's disk requests. For the TPC-C trace, although it takes about 1 h to mine 1 h's trace, it is still practical for storage systems. Because block correlations are relatively stable, it is unnecessary to keep mining for correlations in the background. Instead, it might be acceptable to spend 1 h every week on running *C-Miner* to update correlation rules. In our experiments, we only used parts of the traces to mine correlations, and used the remaining traces to evaluate correlation-directed prefetching and disk layout. Our experimental results indicate that correlations are relatively stable and are useful for accesses made much later after the training period.

C-Miner is also efficient in terms of space overhead for most of traces. It takes less than 10 MB to mine the Cello, TPC-C, and TPC-H traces. With such a small requirement, the data mining can run on the same machine as the storage system without causing too much memory overhead.

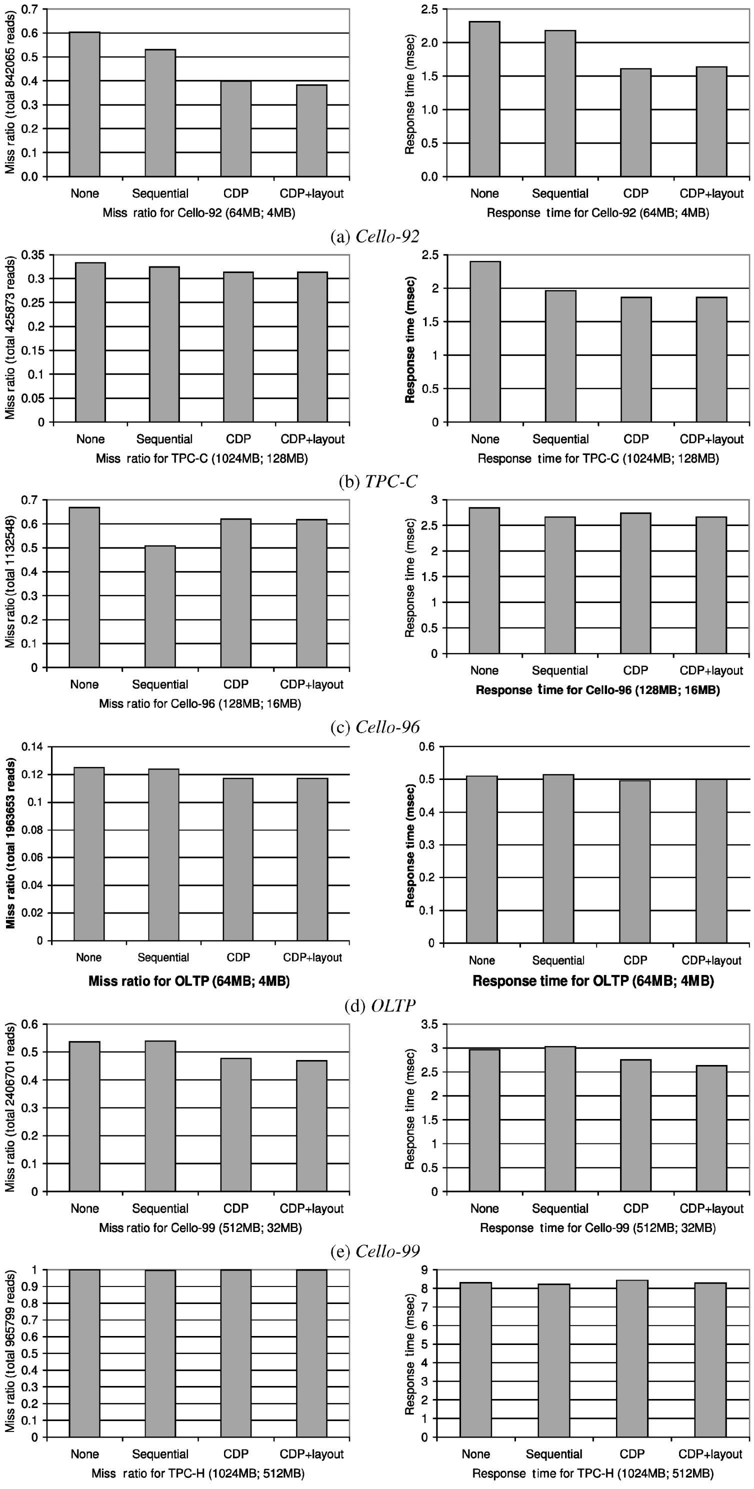
Compared with correlation rules discovered by *C-Miner*, *C-Miner** can discover 7–109% more rules for most of the traces. The results demonstrate that *C-Miner** can find the frequent subsequences from a single long sequence much more than the basic algorithm because *C-Miner* breaks the trace into nonoverlapped short sequences and it may lose patterns. For example, *C-Miner* can find 228,000 rules from Cello-92, while *C-Miner** can discover 335,000— about 50% more than *C-Miner*. Especially for TPC-H, *C-Miner** can discover 12 times more rules than *C-Miner*. The reason is that most frequent sequences in TPC-H have the same support as the threshold *min_sup*, and therefore the loss of one support due to nonoverlapped cutting in *C-Miner* can result in the loss of the frequent sequence.

The results show that *C-Miner** is also as efficient and practical as *C-Miner* in terms of time and space overhead for all traces. Furthermore, *C-Miner** is about 2–15 times faster than *C-Miner* for most of the traces. For example, it only takes less than 9 min for *C-Miner** to mine the 3-days' Cello-92 trace while it takes more than 2 h for *C-Miner*. The exceptional case is TPC-H trace. The reason is that *C-Miner** spends a little longer time discovering 12 times more rules. Although the space overhead of *C-Miner** is larger than *C-Miner*, it is still efficient and practical for the current uniprocessor PCs with 1 GB of memory. The larger space overhead comes from the more frequent subsequences discovered by *C-Miner**.

5.4 Correlation-Directed Prefetching and Disk Layout

In this section, we first present the results of correlation-directed prefetching and disk layout using the advanced algorithm *C-Miner** for all traces, and then compare the results with those using *C-Miner*.

5.4.1 *Results with* C-Miner*. The bar graphs in Figure 9 compare the read miss ratios and response times using the four different schemes: baseline (no-prefetching), sequential prefetching, correlation-directed prefetching (CDP), and correlation-directed prefetching and disk layout (CDP+layout). For the last three schemes with prefetching, the prefetch cache size was set to be the same. All four settings used the same total size of storage cache in order to make a fair comparison. In other words, the *TotalCacheSize*, which equals to the sum of *DemandCacheSize* and *PrefetchCacheSize*, was the same for all four schemes. The rules used in CDP were obtained by *C-Miner**.



(a) *Cello-92*

(b) *TPC-C*

(c) *Cello-96*

(d) *OLTP*

(e) *Cello-99*

(f) *TPC-H*

CDP can improve the average I/O response time for the base-line case by up to 30%. For instance, in Cello-92, CDP had 35.2% lower storage cache hit ratios than the base-line case. This translates into 30.3% improvement in the average I/O response time. In Cello-99, although the miss ratio with CDP was decreased by only 9%, the improvement was quite significant for this trace since such a miss ratio with 512 MB cache with CDP can be achieved by using more than 700 MB cache without CDP. These improvements were due to the fact that prefetching reduces the number of capacity misses as well as the number of cold misses. When the cache size is small, some blocks are evicted and need to be fetched again for disks upon subsequent accesses. Prefetching can avoid misses at some of these accesses.

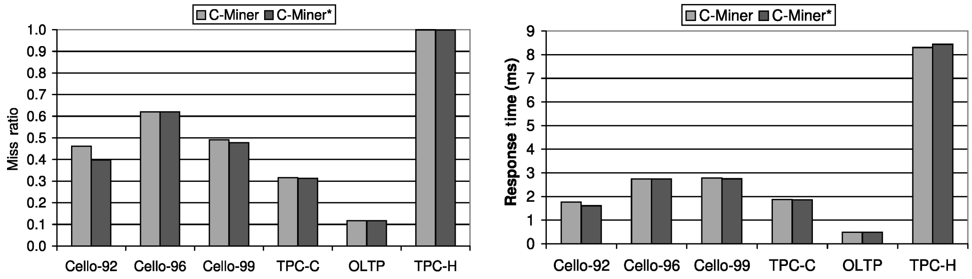
The improvement by CDP was much more significant than that by the commonly used sequential prefetching scheme, especially in the case of TPC-C and Cello-92. For example, for the TPC-C trace, sequential prefetching only slightly reduced the cache miss ratio (by only 2.5%), which was then completely canceled out by the prefetching overheads. Therefore, sequential prefetching had an even worse response time than the base case. For the other two traces (Cello-92 and OLTP trace), the improvement of the sequential prefetching scheme was very small, almost invisible in terms of the average response time. However, in Cello-96, sequential prefetching had a lower miss ratio and slightly better response time than CDP. This was because this trace has a lot of sequential accesses. But these sequential accesses were not frequent enough in the access stream so it was not caught by *C-Miner**. Fortunately, our patterns obtained by *C-Miner** can be complementary and combined with the existing online sequential prefetching algorithms that can detect nonfrequent sequential access patterns.

CDP+layout had only small improvement over CDP. Obviously, CDP+layout should not affect cache miss ratio at all. It only matters to average I/O response time when the disk is heavily utilized. This small improvement indicates that our optimization for hiding prefetching overheads using priority-based disk scheduling is already good enough. Therefore, disk layout does not provide significant benefits. However, when the disk is too heavily utilized for the disk scheduling scheme to hide most of the prefetching overheads, we expect the benefit of correlation-directed disk layout will be larger.

However, no schemes can improve the miss ratio or the response time for TPC-H trace. The reason is that the TPC-H trace is a typical DSS workload and contains a number of sequential accesses. *C-Miner** can only discover the dominant simple block correlations, as we have shown in Figure 7(d). CDP would degenerate to a sequential prefetching scheme using such simple correlations. Furthermore, the database in TPC-H issues a lot of sequential accesses within a large-size request (usually larger than 128 kB). Therefore, the sequential prefetching schemes cannot help improve performance.

←

Fig. 9. Miss ratio and response time. The first number in the parentheses is the total cache size, and the second number is the prefetch cache size. In the base-line case "None," the prefetch cache size is zero, so the demand cache size equals the total cache size. In the other three schemes, the demand cache size is the difference between the total cache size and the prefetch cache size.



(a) *Miss ratio* (b) *Response time*

Fig. 10. Miss ratio and response time with CDP using *C-Miner* and *C-Miner**.

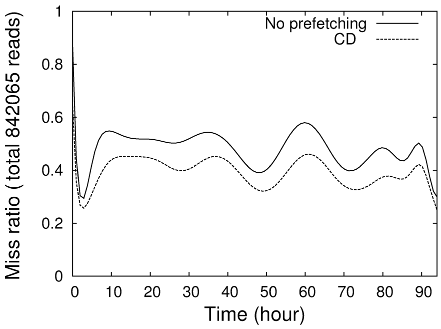
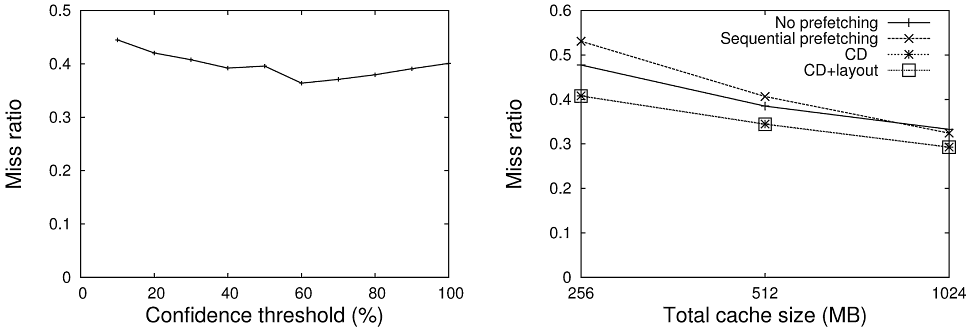


Fig. 11. Miss ratio for Cello-92 (64 MB; 4 MB).

5.4.2 *Comparison of* C-Miner *and* C-Miner* . Because *C-Miner** can find more block correlations than *C-Miner*, as shown in Section 5.3, CDP can perform better using the rules discovered by *C-Miner**. The results of miss ratio and response time of CDP using *C-Miner* and *C-Miner** are compared in Figure 10. With the more block correlations discovered by *C-Miner**, CDP scheme can prefetch more data and therefore improve the average response time. For example, because *C-Miner** can find 50% more rules from Cello-92 than *C-Miner* , the miss ratio of CDP with *C-Miner** can be improved by 30%, while the miss ratio of CDP with *C-Miner* can be improved by 25%. For Cello-96 and TPC-C, because not many more rules can be found by *C-Miner**, the miss ratio and response time are very similar using *C-Miner* and *C-Miner**.

5.5 Stability of Block Correlations

In order to show that block correlations are relatively stable, we use the correlation rules mined from the *first 3 days* of the Cello-92 trace using *C-Miner**. Our simulator applies these rules to the next 4 days' trace without updating any rules. Figure 11 shows the miss ratio for the *next 4 days'* trace using correlated-directed prefetching (CDP). The miss ratios in the figure are calculated by aggregating every 10,000 read operations. This figure shows that CDP is always better than the base case. This implies that correlations mined from the first



(a) *Effect of the confidence threshold for Cello-92*

(b) *Effect of varying the total cache size for TPC-C*

Fig. 12. Impact of configuration.

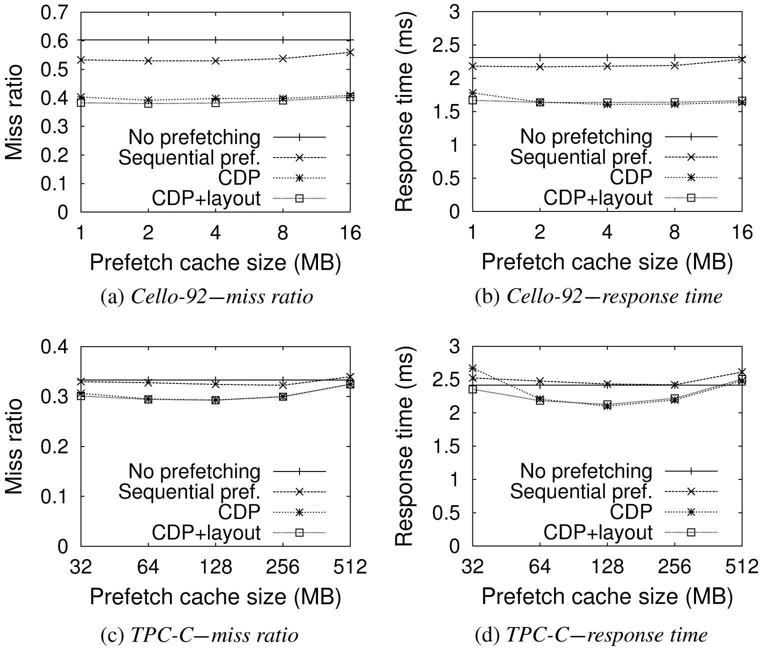
3 days are still effective for the next 4 days. In other words, block correlations are relatively stable for a relative long period of time. Therefore, there is no need to run *C-Miner** (or *C-Miner*) continuously in the background to update block correlations. This also shows that, as long as the mining algorithm is reasonably efficient, the mining overhead is not a big issue.

5.6 Impact of Configurations

5.6.1 *Effects of the Confidence Threshold.* A parameter that can affect the benefits of correlation directed prefetching is the confidence threshold of correlation rules. Figure 12(a) shows the effects of varying the confidence threshold from 10% to 100%. A lower confidence threshold corresponds to a more aggressive prefetching policy. The figure shows that the miss ratio is minimum when prefetching is moderate and the rules with confidence above 60% are used. We can see that the miss ratio increases when prefetching is either too conservative or too aggressive (when the rule confidence is smaller than 50%). The reason for aggressive prefetching is that, when the rules with low confidence are used, some mispredicted blocks may pollute the prefetch cache.

5.6.2 *Effects of the Total Cache Sizes.* If the cache size is comparable to the footprint of a trace, the system simply caches all accesses. Because of this, the read misses in the case of no prefetching are predominantly cold misses since subsequent accesses will be cache hits and will not go to the disk. Therefore, the prefetching schemes do not yield much improvement in performance.

We study the effects of the cache size by varying the cache size for TPC-C exponentially from 256 MB to 1024 MB, as shown in Figure 12(b). In this experiment, we keep the prefetch cache fixed at 128 MB and vary the size of the demand cache. As expected, when the cache size is set at 256 MB, CDP+layout shows an improvement of 14.6% in miss ratio while with 512 MB, the improvement is only 10.6%. It is important to note that our workloads have relatively small working set sizes. In large real systems, it is usually not the case that the entire working set can fit into main memory.



(a) *Cello-92—miss ratio*

(b) *Cello-92—response time*

(c) *TPC-C—miss ratio*

(d) *TPC-C—response time*

Fig. 13. Effect of varying the prefetch cache size. (Note: the total cache size is fixed. Therefore, the demand cache size is also changing with the prefetch cache size. In the base-line case, there is no prefetch cache.)

5.6.3 *Effects of the Prefetch Cache Size.* Figures 13(a)–13(d) show the effects of varying the prefetch cache size while the total cache size is fixed. In TPC-C, for instance, the storage cache miss ratio with CDP initially decreases as the prefetch cache size increases. It reaches the minimum when the prefetch cache size is set to 128 MB. Beyond that point the miss ratio increases again with the prefetch cache size. This phenomenon is also true for the sequential prefetching, even though it performs worse than CDP with almost all prefetch cache sizes.

The above phenomenon can be quite expected. When the prefetch cache size is very small, prefetched blocks may be replaced even before they are used. Even though the demand cache size is increased correspondingly, its benefit is not large enough to offset the loss in unused prefetches. Even worse, the overhead imposed by CDP causes an increase in the response time compared to the base case, as shown on Figure 13(d). Fortunately, in this case, the CDP+layout starts to show the benefits of correlation-directed disk layout. It is still able to provide some small improvement over the base line case.

As the prefetch cache size increases, blocks can be retained longer in the prefetch cache and subsequently be used to handle block requests. However, the increase in prefetch cache size corresponds to a reduced demand cache size, but the benefit of prefetching in reducing misses outweighs the loss in the demand cache. Beyond 128 MB, increasing the prefetch cache size no longer has benefits for increasing hit ratio. So the loss due to the reduced demand cache size starts to dominate. Therefore, the overall miss ratio increases.

6. DISCUSSION

In order to use *C-Miner* and *C-Miner** in online storage systems, some issues should be addressed.

The first issue is how the algorithms can be run in such systems. One way is to run them on the same system in the background during the idle period. The running time and space overhead results shown in Section 5.3 indicate that our mining algorithms are practical to run on the modern storage systems. To reduce the overhead for collecting I/O traces, a specific log disk can be used. In order to avoid the performance degradation for the original system, another way is to run the algorithms on a separate machine where a PC is powerful enough.

The second issue is how to update block correlations. Since block correlations depend on data semantics, they can remain relatively stable for a long time in most systems. Therefore, it is unnecessary to run the algorithms continuously. The system can collect I/O traces and feed them to the mining algorithm periodically or when the block correlations are detected to be out of date (for example, when the hit ratio of prefetching in CDP is too low).

Furthermore, an incremental stream mining algorithms would be more desirable for updating block correlations. With such enhanced algorithms, we do not need to keep the traces any more because all the I/O accesses can be directly passed to the mining algorithm as a data stream online. Additionally, to consider recency of correlations (that is, some recent frequent subsequences may be more important than the "ancient" ones), the stream can be considered as time-series data stream. Accordingly, the mining algorithms can be modified to change the support or confidence value of a rule dynamically as time progresses, and so the "ancient" patterns that have not appeared for a long time would be removed from the correlation rule set.

7. RELATED WORK

In this section, we briefly discuss some representative work that is closely related to our work. Section 2 has discussed various approaches to capture data semantics, and we do not repeat them here.

Data prefetching has also been studied extensively in databases, file systems, and parallel applications with intensive I/Os. Most of previous prefetching work either relies on applications to pass hints or is based on simple heuristics such as sequential accesses. Examples of prefetching studies for databases include Smith [1978b], Wedekind and Zoerntlein [1986], Palmer and Zdonik [1991], Gerlhof and Kemper [1994a, 1994b], and Hsu et al. [2001] as well as some recent work [Seifert and Scholl 2002] for mobile data delivery environments. Prefetching for file I/Os includes application-controlled prefetching [Cao et al. 1994, 1995] and informed prefetching [Tomkins et al. 1997; Kimbrel et al. 1996; Patterson et al. 1995], just to name a few. Soloviev [1996] is an example of prefetching in disk caches. I/O prefetching for out-of-core applications includes compiler-assisted prefetching [Mowry et al. 1996; Brown et al. 2001] and prefetching through speculative execution [Chang and Gibson 1999].

In the spectrum of sophisticated prefetching schemes, research has been conducted for semantic distance-based file prefetching for mobile or networked file servers. Besides the probability graph-based approach described in Section 2, the SEER project from UCLA [Kuenning 1994; Kuenning and Popek 1997] groups related files into clusters by keeping track of semantic distances between files and downloading as many complete clusters as possible onto the mobile station. The CLUMP project tries to leverage the concept of semantic distance to prefetch file clusters [Eaton et al. 1999]. Kroeger extended the probability graph to a trie with each node representing the sequence of consecutive file accesses from the root to the node [Kroeger and Long 1995]. Lei and Duchamp also used a similar structure by building a probability tree [Tait et al. 1995; Lei and Duchamp 1997]. Vellanki and Chervenak [1999] combined Patterson's cost-benefit analysis with probabilistic prefetching for high performance parallel file systems. Similar to the probability graph, most of these approaches may be feasible for prefetching at file granularity, but are impractical to track block correlations in a storage system (see Section 2).

Some studies used data compression techniques for prefetching. It was first proposed by Vitter and Krishnan [1991]. The basic idea is to encode the data expected with higher probability using fewer bits. The prefetchers based on any optimal character-by-character data compressor were theoretically proven to be optimal in page fault rate. Later, Curewitz et al. [1993] analyzed some practical issues of such a technique, and proposed three practical data compressors for prefetching.

Data mining methods have been mostly used to discover patterns in sales, finance or bio-informatics databases [Han and Kamber 2001; Han 2002]. Only a few studies have applied them in systems. A well-known example is using data mining for intrusion detection [Lee and Stolfo 1998; Clifton and Gengo 2000]. Data mining has recently been used in performance evaluation [Wang et al. 2002] to model bursty traffic.

Data mining and machine learning have been used in web environments to predict HTTP requests. Schechter et al. [1998] introduced path profiling to predict HTTP requests in Web environments. Pitkow and Pirolli [1999] have used longest repeating subsequences to perform path matching for predicting Web accesses from a client. These schemes predict the next HTTP request by matching the surfer's current sequence against the path profile database.

While path-based prediction may work very well for Web environments, it is very difficult to capture block correlations in storage systems. This is because Web browser/server workloads are different from storage workloads. Each Web client usually only browses one page at a time, whereas a storage front-end such as database server can have hundreds of outstanding requests. Since the path-matching schemes do not allow any gaps in the subsequence or path, they cannot be used easily to capture block correlations in a storage system. Supporting gaps or lookahead distances in these approaches will suffer the same problem as encountered in the probability graph-based approach.

Our work is also related to various adaptive approaches using learning techniques [Madhyastha and Reed 1997; Madhyastha et al. 1999; Ari et al. 2002;

Madhyastha and Reed 2002], intelligent storage cache management [Zhou et al. 2001; Wong and Wilkes 2002; Megiddo and Modha 2003; Chen et al. 2003], and autonomic storage systems [Wilkes et al. 1995; Anderson et al. 2002; Keeton and Wilkes 2002]

8. CONCLUSIONS AND FUTURE WORK

This article first proposes *C-Miner*, a novel algorithm that uses data mining techniques to systematically mine access sequences in a storage system to infer block correlations. In order to increase the accuracy and time efficiency of *C-Miner*, we further propose a new algorithm, called *C-Miner**, for mining correlations from a single long sequence such as the storage system traces without cutting. Using several large real system disk traces, our experiments show that both algorithms are reasonably fast with small space requirement and therefore practical to be used online in autonomic storage systems.

We have also evaluated correlation-directed prefetching and data layout. Our experimental results with real-system traces have shown that correlation-directed prefetching and data layout can improve I/O average response time by 12–30% compared to no-prefetching, and 7–25% compared to the commonly used sequential prefetching for most workloads.

Our study has several limitations. First, even though this article focuses on how to obtain block correlations, our evaluation of the block correlation-directed prefetching and disk layout was conducted using only simulations. We are in the process of implementing correlation-directed prefetching and disk layout in our previously built storage system. Second, we do not compare with the semantic-distance graph approach. The main reason is that our preliminary experiment indicates that the SD graphs significantly exceed the memory space, making it extremely slow and almost infeasible to build such graphs.

As we discussed in Section 6, the online and incremental mining algorithm is desirable for deploying our technique in real systems. Currently, we are designing efficient stream data mining algorithms specifically for mining access sequences for storage systems and any other application scenarios.

ACKNOWLEDGMENTS

We appreciate Kimberly Keeton from HP labs for the constructive discussion and thank HP Storage System Labs for providing us Cello traces. We are also grateful to Professor Jiawei Han and his student Xifeng Yan for their help with the CloSpan algorithm and insightful discussions.

REFERENCES

AGRAWAL, R. AND SRIKANT, R. 1995. Mining sequential patterns. In *Proceedings of the Eleventh International Conference on Data Engineering*.

ANDERSON, E., HOBBS, M., KEETON, K., SPENCE, S., UYSAL, M., AND VEITCH, A. 2002. Hippodrome: Running circles around storage administration. In *Proceedings of the First USENIX Conference on File and Storage Technologies*.

ANTHES, G. H. 2002. Storage virtualization: The next step. *Computer World*, January 28, 2002, p. 43.

ARI, I., AMER, A., MILLER, E., BRANDT, S., AND LONG, D. 2002. Who is more adaptive? ACME: Adaptive caching using multiple experts. In *Proceedings of the Workshop on Distributed Data and Structures* (WDAS).

ARPACI-DUSSEAU, A. C. AND ARPACI-DUSSEAU, R. H. 2001. Information and control in gray-box systems. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*.

AYRES, J., FLANNICK, J., GEHRKE, J., AND YIU, T. 2002. Sequential pattern mining using a bitmap representation. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM Press, New York, NY, 429–435.

BARROSO, L. A., GHARACHORLOO, K., AND BUGNION, E. 1998. Memory system characterization of commercial workloads. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*. IEEE Press, Los Alamitos, CA, 3–14.

BROWN, A. D., MOWRY, T. C., AND KRIEGER, O. 2001. Compiler-based I/O prefetching for out-of-core applications. *ACM Trans. Comput. Syst. 19*, 2, 111–170.

CAO, P., FELTEN, E., AND LI, K. 1994. Application-controlled file caching policies. In *Proceedings of the USENIX Summer 1994 Technical Conference*. 171–182.

CAO, P., FELTEN, E. W., KARLIN, A., AND LI, K. 1995. A study of integrated prefetching and caching strategies. In *Proceedings of ACM SIGMETRICS*.

CARRERA, E. V., PINHEIRO, E., AND BIANCHINI, R. 2003. Conserving disk energy in network servers. In *Proceedings of the 17th International Conference on Supercomputing*.

CHANG, F. W. AND GIBSON, G. A. 1999. Automatic I/O hint generation through speculative execution. In *Proceedings of the Conference on 2003 Operating Systems Design and Implementation*. 1–14.

CHEN, Z., ZHOU, Y., AND LI, K. 2003. Eviction-based cache placement for storage caches. In *Proceedings of the Conference on 2003 USENIX Annual Technical Conference*. 269–282.

CHOI, J., NOH, S. H., MIN, S. L., AND CHO, Y. 2000. Towards application/file-level characterization of block references: A case for fine-grained buffer management. In *Proceedings of the 2000 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*.

CHOU, H. AND DEWITT, D. 1993. An evaluation of buffer management strategies for relational database systems. In *Proceedings of the 19th International Conference on Very Large Data Bases* (Dublin, Ireland). 127–141.

CLIFTON, C. AND GENGO, G. 2000. Developing custom intrusion detection filters using data mining. In *Proceedings of the 2000 Military Communications International Symposium* (MILCOM2000, Los Angeles, CA).

CUREWITZ, K. M., KRISHNAN, P., AND VITTER, J. S. 1993. Practical prefetching via data compression. In *Proceedings of the 1993 ACM-SIGMOD Conference on Management of Data*. 257–266.

EATON, P. R., GEELS, D., AND MORI, G. 1999. Clump: Improving file system performance through adaptive optimizations. Go to http://www.citeseer.csail.mit.edu/eatox99clump.html.

EMC CORPORATION. 1999. *Symmetrix 3000 and 5000 Enterprise Storage Systems Product Description Guide*. EMC Corporation, Hopkinton, MA. Web site: http://www.emc.com.

GANGER, G. 1995. Systemoriented evaluation of I/O subsystem performance. Tech. rep. CSE-TR-243-95. University of Michigan, Ann Arbor, MI.

GERLHOF, C. A. AND KEMPER, A. 1994a. A multi-threaded architecture for prefetching in object bases. In *Advances in Database Technology—EDBT'94. 4th International Conference on Extending Database Technology, Cambridge, United Kingdom, March 28-31, 1994, Proceedings*, M. Jarke, J. A. B. Jr., and K. G. Jeffery, Eds. Lecture Notes in Computer Science, vol. 779. Springer, Berlin, Germany, 351–364.

GERLHOF, C. A. AND KEMPER, A. 1994b. Prefetch support relations in object bases. In *Persistent Object Systems, Proceedings of the Sixth International Workshop on Persistent Object Systems, Tarascon, Provence, France, 5–9 September 1994*, M. P. Atkinson, D. Maier, and V. Benzaken, Eds. Workshops in Computing. Springer, Berlin, Germany, and British Computer Society, Swindon, Wilts., U.K., 115–126.

GIBSON, G. A., NAGLE, D. F., AMIRI, K., BUTLER, J., CHANG, F. W., GOBIOFF, H., HARDIN, C., RIEDEL, E., ROCHBERG, D., AND ZELENKA, J. 1998. A cost-effective, high-bandwidth storage architecture. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS).

GRIFFIOEN, J. AND APPLETON, R. 1994. Reducing file system latency using a predictive approach. In *Proceedings of the 1994 Summer USENIX Conference*.

GRIFFIOEN, J. AND APPLETON, R. 1995. Performance measurements of automatic prefetching. In *Proceedings of the International Conference on Parallel and Distributed Computing Systems*.

HAN, J. 2002. How can data mining help bio-data analysis? In *Proceedings of the 2002 Workshop on Data Mining in Bioinformatics* (BIOKDD'02, Edmonton, Canada). 1–4.

HAN, J. AND KAMBER, M. 2001. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers, San Francisco, CA.

HSU, W. W., SMITH, A. J., AND YOUNG, H. C. 2001. I/O reference behavior of production database workloads and the TPC benchmarks—an analysis at the logical level. *ACM Trans. Database Syst. 26*, 1, 96–143.

IBM. 2002. Storage Tank, a distributed storage system. IBM White paper. Web site: `http://www.almaden.ibm.com/StorageSystems/file_systems/storage_tank/papers.shtml`.

KEETON, K. AND WILKES, J. 2002. Automating data dependability. In *Proceedings of 10th ACM-SIGOPS European Workshop.*

KIM, J., CHOI, J., KIM, J., NOH, S., MIN, S., CHO, Y., AND KIM, C. 2000. A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation* (OSDI, San Diego, CA). 119–134.

KIMBREL, T., TOMKINS, A., PATTERSON, R. H., BERSHAD, B., CAO, P., FELTEN, E., GIBSON, G., KARLIN, A. R., AND LI, K. 1996. A trace-driven comparison of algorithms for parallel prefetching and caching. In *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation*. USENIX Association, Berkeley, CA, 19–34.

KROEGER, T. M. AND LONG, D. D. E. 1995. Predicting file-system actions from prior events. In *Proceedings of the 1996 USENIX Annual Technical Conference*. 319–328.

KUENNING, G. 1994. Design of the SEER predictive caching scheme. In *Proceedings of the Workshop on Mobile Computing Systems and Applications*.

KUENNING, G. H. AND POPEK, G. J. 1997. Automated hoarding for mobile computers. In *Proceedings of the 15th Symposium on Operating Systems Principles* (St. Malo, France). ACM Press, New York, NY, 264–275.

LEE, E. K. AND THEKKATH, C. A. 1996. Petal: Distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM Press, New York, NY, 84–92.

LEE, W. AND STOLFO, S. 1998. Data mining approaches for intrusion detection. In *Proceedings of the 7th USENIX Security Symposium* (San Antonio, TX).

LEI, H. AND DUCHAMP, D. 1997. An analytical approach to file prefetching. In *Proceedings of the 1997 USENIX Annual Technical Conference* (Anaheim, CA.)

LEUTENEGGER, S. T. AND DIAS, D. 1993. A modeling study of the TPC-C benchmark. *SIGMOD Rec. 22*, 2 (June), 22–31.

MADHYASTHA, T. M., GIBSON, G. A., AND FALOUTSOS, C. 1999. Informed prefetching of collective input/output requests. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing* (CDROM). ACM Press, New York, NY, 13.

MADHYASTHA, T. M. AND REED, D. A. 1997. Input/output access pattern classification using hidden Markov models. In *IOPADS '97: Proceedings of the Fifth Workshop on I/O in Parallel and Distributed Systems*. ACM Press, New York, NY, 57–67.

MADHYASTHA, T. M. AND REED, D. A. 2002. Learning to classify parallel input/output access patterns. *IEEE Trans. Parallel Distrib. Syst. 13*, 8, 802–813.

MEGIDDO, N. AND MODHA, D. S. 2003. Arc: A self-tuning, low overhead replacement cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies* (FAST 03, San Francisco, CA).

MOWRY, T. C., DEMKE, A. K., AND KRIEGER, O. 1996. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation*. USENIX Association, Berkeley, CA, 3–17.

PALMER, M. AND ZDONIK, S. B. 1991. Fido: A cache that learns to fetch. In *17th International Conference on Very Large Data Bases, September 3–6, 1991, Barcelona, Catalonia, Spain,* *Proceedings*, G. M. Lohman, A. Sernadas, and R. Camps, Eds. Morgan Kaufmann, San Francisco, CA, 255–264.

Patterson, R. H., Gibson, G. A., Ginting, E., Stodolsky, D., and Zelenka, J. 1995. Informed prefetching and caching. In *Proceedings of the 15th SOSP*.

Pei, J., Han, J., Mortazavi-Asl, B., Pinto, H., Chen, Q., Dayal, U., and Hsu, M.-C. 2001. PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *Proceedings of the 2001 International Conference on Data Engineering* (ICDE'01, Heidelberg, Germany). 215–224.

Pitkow, J. E. and Pirolli, P. 1999. Mining longest repeating subsequences to predict World Wide Web surfing. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*.

Ruemmler, C. and Wilkes, J. 1993a. A trace-driven analysis of disk working set sizes. Tech. rep. HPL-OSR-93-23. Hewlett-Packard Laboratories, Palo Alto, CA.

Ruemmler, C. and Wilkes, J. 1993b. UNIX disk access patterns. In *Proceedings of the Winter 1993 USENIX Conference*.

Salmon, B., Thereska, E., Soules, C. A., and Ganger, G. R. 2003. A two-tiered software architecture for automated tuning of disk layouts. In *Proceedings of the First Workshop on Algorithms and Architectures for Self-Managing Systems*.

Schechter, S., Krishnan, M., and Smith, M. D. 1998. Using path profiles to predict http requests. In *Proceedings of the Seventh International World Wide Web Conference*.

Schindler, J., Griffin, J., Lumb, C., and Ganger, G. 2002. Track-aligned extents: Matching access patterns to disk drive characteristics. In *Proceedings of the First USENIX Conference on File and Storage Technologies*.

Seifert, A. and Scholl, M. H. 2002. A multi-version cache replacement and prefetching policy for hybrid data delivery environments. In *Proceedings of the 28th International Conference on Very Large Data Bases* (VLDB).

Sivathanu, M., Prabhakaran, V., Popovici, F., Denehy, T. E., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. 2003. Semantically-smart disk systems. In *Proceedings of the Second USENIX Conference on File and Storage Technologies*.

Smith, A. J. 1978a. Sequentiality and prefetching in database systems. *ACM Trans. Database Syst. 3*, 3 (Sept.), 223–247.

Smith, B. J. 1978b. A pipelined, shared resource MIMD computer. In *Proceedings of International Conference on Parallel Processing*. 6–8.

Soloviev, V. 1996. Prefetching in segmented disk cache for multi-disk systems. In *Proceedings of the Fourth Workshop on I/O in Parallel and Distributed Systems*. ACM Press, New York, NY, 69–82.

Storage Performance Council. 2004. SPC I/O traces. Web site: `http://www.storageperformance.org/`.

Tait, C. D., Lei, H., Acharya, S., and Chang, H. 1995. Intelligent file hoarding for mobile computers. In *Proceedings of the Conference on Mobile Computing and Networking*. 119–125.

Tomkins, A., Patterson, R. H., and Gibson, G. 1997. Informed multi-process prefetching and caching. In *Proceedings of the 1997 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. ACM Press, New York, NY, 100–114.

Vellanki, V. and Chervenak, A. 1999. A cost-benefit scheme for high performance predictive prefetching. In *Proceedings of SC99: High Performance Networking and Computing* (Portland, OR). ACM Press, New York, NY, and IEEE Computer Society Press, Los Alamitos, CA.

Vitter, J. S. and Krishnan, P. 1991. Optimal prefetching via data compression. In *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science*.

Wang, M., Madhyastha, T., Chan, N. H., Papadimitriou, S., and Faloutsos, C. 2002. Data mining meets performance evaluation: Fast algorithms for modeling bursty traffic. In *Proceedings of the 18th International Conference on Data Engineering*.

Wedekind, H. and Zoerntlein, G. 1986. Prefetching in realtime database applications. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*. ACM Press, New York, NY, 215–226.

Wilkes, J., Golding, R., Staelin, C., and Sullivan, T. 1995. The HP AutoRAID hierarchical storage system. In *Proceedings of the 15th Symposium on Operating Systems Principles*.

WONG, T. AND WILKES, J. 2002. My cache or yours? Making storage more exclusive. In *Proceedings of USENIX*.

YAN, X., HAN, J., AND AFSHAR, R. 2003. CloSpan: Mining closed sequential patterns in large datasets. In *Proceedings of the 2003 SIAM International Conference Data Mining* (SDM'03, San Francisco, CA).

ZAKI, M. 2001. SPADE: An efficient algorithm for mining frequent sequences. *Machine Learn. 40*, 31–60.

ZHANG, Y., ZHANG, J., SIVASUBRAMANIAM, A., LIU, C., AND FRANKE, H. 2003. Decision-support workload characteristics on clustered database server from the OS perspective. In *Proceedings of the International Conference on Distributed Conputing Systems* (ICDCS).

ZHOU, Y., PHILBIN, J. F., AND LI, K. 2001. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of the Usenix Technical Conference*.

Received September 2004; revised December 2004; accepted December 2004