I/O Characteristic Discovery for Storage System Optimizations

Jiang Zhou[a], Yong Chen[b,*], Dong Dai[c], Yu Zhuang[b], Weiping Wang[a]

[a]*Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China*
[b]*Department of Computer Science, Texas Tech University, Lubbock, USA*
[c]*Department of Computer Science, University of North Carolina at Charlotte, Charlotte, USA*

Abstract

In this paper, we introduce a new I/O characteristic discovery methodology for performance optimizations on object-based storage systems. Different from traditional methods that select limited access attributes or heavily reply on domain knowledge about applications' I/O behaviors, our method enables capturing data-access features as many as possible to eliminate human bias. It utilizes a machine-learning based strategy (principal component analysis, PCA) to derive the most important set of features automatically, and groups data objects with a clustering algorithm (DBSCAN) to reveal I/O characteristics discovered. We have evaluated the proposed I/O characteristic discovery solution based on Sheepdog storage system and further implemented a data prefetching mechanism as a sample use case of this approach. Evaluation results confirm that the proposed solution can successfully identify access patterns and achieve efficient data prefetching by improving the buffer cache hit ratio up to 48.24%. The overall performance was improved by up to 42%.

Keywords: Parallel/distributed file systems, object-based storage, I/O characteristic discovery, access pattern analysis, I/O optimization

*Corresponding author
Email addresses: zhoujiang@iie.ac.cn (Jiang Zhou), yong.chen@ttu.edu (Yong Chen), dong.dai@uncc.edu (Dong Dai), yu.zhuang@ttu.edu (Yu Zhuang), wangweiping@iie.ac.cn (Weiping Wang)

1. Introduction

The ever-increasing data demand in many science and engineering domains has posed significant challenges to storage systems. Parallel/distributed object-based store [1], such as Lustre [2], Ceph [3], and Sheepdog [4], has been widely used to provide required storage capability, and more importantly, to deliver high data-accesses speed.

To achieve highly optimized data-access performance of storage systems, understanding and leveraging data-access patterns have been proven effective. For example, PDLA [5] describes an I/O data replication scheme that replicates identified data-access pattern (i.e., spatial-related access pattern), and saves these reorganized replications with optimized data layouts based on access cost analysis. Arguably, the better the access pattern is understood, the better the storage system can be optimized and tuned.

As a result, numerous studies have been conducted to identify, characterize, and leverage data-access patterns in storage systems. One well-known method is to analyze spatial/temporal behaviors of data accesses to identify I/O access sequence [6, 7, 8, 9, 10]. Another important method is to analyze semantic information such as the access correlations between file blocks/objects by mining I/O semantic attributes, which can potentially discover more complex patterns, especially semantic patterns [11, 12, 13, 14, 15]. Other methods also exist at different I/O layers, such as application-/client-side trace analysis [7], sever-side trace analysis [16, 17], or both server and client I/O analysis [11]. While existing studies show the feasibility of I/O characteristic discovery through various approaches, they have three shortcomings as discussed below, which also motivates this research.

First, many of these approaches are limited to specific and predefined features. A feature refers to an attribute of a data access. For example, the object ID of an object access is a feature; the data access size is another feature. Many existing studies investigate one or more rather specific, predefined features to analyze access patterns. The resulting insights, although valuable, often lead to an incomplete view of access patterns. For example, *object ID* and *access time* help obtain temporal I/O behavior, but, if *access length* and *offset* are considered, spatial correlation of data accesses can be further derived. In general, more features indicate more I/O behaviors, i.e., read/write

operation code reveals read/write types and *target node ID* provides object location. Given the increasing complexity of I/O behaviors, it is inadequate to attempt to discover I/O characteristics with only specific or predefined features. It is critical to analyze abundant features thoroughly for I/O characteristics discovery.

Second, existing approaches often introduce bias. Clearly, treating all features equally is not accurate because distinct features can have significantly different impacts on I/O characterization. However, selecting the desired set of features is often daunting and introduces bias, which requires domain knowledge and assumptions about storage systems and applications. Besides, we will not know whether we have a desired set of features until we have completed the entire analysis process. For instance, the authors in study [11] presented an iterative process to initially select some basic features, e.g., total I/O size and read-write ratio for a file, and then add new features if the analysis results leave some system design choice ambiguous. They still need to interpret the output results and derive access patterns by looking at only the relevant subset of features, again using domain knowledge. Moreover, the system may exhibit various data-access patterns for a given application. For example, the scientific computing applications in study [18] show various data-access patterns for massive data processing. Identifying representative features manually would not adapt to this scenario, and may lead to an untenable analysis. This drawback largely limits the efficiency of using access patterns for tuning storage system and optimizing the performance.

Third, existing approaches often have limited adaptability and have constraints on utilizing discovered I/O characteristics. They usually focus on either mining spatial/temporal patterns of applications for I/O acceleration, or exploring more complex patterns for data/block access optimizations. However, I/O characteristics are very useful in many other scenarios too, such as in storage system tuning, data prefetching, data placement, data organization, etc. A comprehensive, holistic design on discovering I/O characteristics and optimizing storage systems is strongly desired.

In this paper, we present a new design methodology to overcome these shortcomings of existing methods discussed above. Specifically, we propose to capture features of data accesses on object-based storage systems as many as possible (more than 20 in our test cases), including features like *object ID*, *access time*, *target node*, and oth-

ers we can collect, to generalize data-access pattern discovery for various applications. Based on the rich set of features, we use access correlations among objects to identify different patterns. We utilize machine-learning based strategy (principal component analysis, PCA [19]) to find the most important "key features" automatically among all collected features in an unsupervised way. This eliminates the bias from users or domain knowledge requires for applications, and provides an automatic, extensible way to identify the dominant data-access patterns. Based on the learned key features, we apply a clustering algorithm (i.e., DBSCAN) to mine the objects' I/O similarity, particularly "key feature correlations", and group highly relevant object IDs, which can be leveraged to improve the I/O performance.

We have implemented the data-access pattern discovery and the prefetching mechanism based on the identified patterns on Sheepdog, a production object-based store. We evaluated the performance benefits using standard benchmarks and synthetic workloads. Our results confirm that our proposed solution can accurately identify the data-access patterns under various workloads and the prefetching strategy can efficiently leverage pattern analysis results to improve the read cache hit ratio (up to 48.24%) and the overall performance (up to 42%). These proof-of-concept evaluations indicate that our I/O characteristic discovery methodology is highly promising for I/O tuning and optimizations. A preliminary study of this research appear in [20]. The key contributions of this research are five-fold:

- Introduce a new I/O characteristics discovery and data-access pattern analysis strategy based on a large number of collectable features of I/O accesses for object-based storage systems.
- Utilize machine-learning based techniques to identify key features for I/O accesses and to cluster objects by mining objects' I/O similarity for I/O performance optimizations.
- Eliminate human bias in discovering I/O patterns without requiring domain knowledge about applications' I/O behaviors.
- Design and implement a data prefetcher prototype based on I/O characteristic

discovery on a real storage system.

- Conduct extensive evaluations on Sheepdog system as a case study to validate the proposed I/O characteristic discovery solution.

The rest of this paper is organized as follows. Section 2 introduces the background of object storage and I/O trace. Section 3 describes the design and implementation of I/O characteristic discovery methodology. Section 4 describes the use case of prefetching by leveraging access patterns for performance improvement. Section 5 presents extensive evaluation results. Section 6 reviews the related work and Section 7 concludes this paper.

2. Object Storage and I/O Trace

In this section, we first introduce the generic I/O software stack of object storage systems that our proposed characteristic discovery methods are based upon, and discuss how I/O accesses can be captured. Then, based on a production-level object-based storage system, Sheepdog [4], we further describe the trace collection mechanism, designed and implemented as the basis for this study.

2.1. Object-based Storage and I/O Trace

In most object-based storage models, applications access data files through the POSIX interface. Internally, data files are abstracted as multiple fixed-size data objects and stored in object-based devices. Under such model, three I/O software stack layers can be identified [21, 22]. The top layer runs on compute nodes (i.e., clients) within user applications, where the I/O accesses are issued from. The second layer includes both the client-side file system libraries and runtime supports. The object-based storage systems need to implement the functions of file read and write operations to support the POSIX interface. Data accesses will be mapped to objects in this layer. At the third layer, these mapped object requests will be dispatched to different storage nodes for accessing data.

Data accesses can be traced at any I/O stack layer. For example, Darshan [23] collects I/O trace statistics at the first layer via instrumenting I/O calls made by applications. In this study, we focus on tracing object accesses at the third layer, i.e., on the storage nodes (server-side). There are mainly two reasons for this design decision. First, all first- and second-layer I/O behaviors will be ultimately turned into object accesses at the third layer. I/O accesses can be fully collected through tracing on the server side. Second, tracing server-side accesses does not assume any domain knowledge or priori information of applications, which is critical to support a system-level I/O characteristics discovery methodology for applications.

2.2. I/O Trace Collection

As discussed earlier, tracing object accesses on the storage node side presents a global and complete view of I/O accesses of an entire storage system. Hence, we implement our I/O trace infrastructure at this layer. Note that, the implementation of such a tracing mechanism will be highly dependent on the architecture of the object-based storage. Some object-based storage systems utilize a centralized architecture, where all I/O requests will go through some *system servers* (or metadata servers). Therefore, tracing these system servers would be sufficient [2, 24]. On the other hand, some other storage systems take a decentralized design, where I/O requests are directly sent to the storage nodes (e.g., via consistent hashing algorithm [4]). Although this design complicates the implementation of tracing as multiple concurrent tracing instances need to run on all storage servers, collecting object access traces in a decentralized architecture is still feasible and practical.

In this study, we focus on a distributed object-based storage system, Sheepdog, which follows a decentralized design, but at the same time, assigns some storage servers as the *gateway nodes* to receive or forward all I/O requests. We call the *gateway nodes* as *aggregators* and use them to design and implement the server-side object access tracing mechanism.

Figure 1 shows the I/O trace collection on Sheepdog system. Sheepdog provides block-level storage volumes attached to QEMU/KVM virtual machines (VMs). Applications run on VMs that are resident on physical machines. For each VM, the applica-

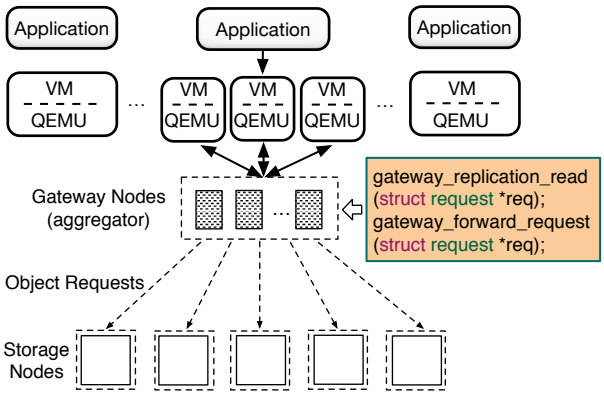


Figure 1: I/O trace collection on Sheepdog object-based storage system

tion I/O is transformed to read/write operations on virtual disk image file, which is split into objects and stored on Sheepdog. Although there are numerous tools available to trace I/O activity [25, 26], we embed codes on the implementation of *gateway nodes*, in functions *gateway_replication_read* and *gateway_forward_request*, for lightweight trace collection. For each *gateway node*, the trace data is flushed to a trace file at the background. If there are more than one *gateway node* configured, multiple trace files are merged before performing pattern analysis. By analyzing I/O trace on the aggregator, I/O characteristics can be mined for the entire storage system. The tracing and I/O characteristics discovery methodologies can also be applied to general parallel/distributed storage systems on physical nodes.

3. I/O Characteristic Discovery

In this section, we introduce the design and implementation of the proposed I/O characteristic discovery methodology. Figure 2 shows an overview of this method. It begins with a stream of I/O traces, which can be collected periodically based on the description in Section 2.2. Each line/record in the trace file indicates one object access with various features. Figure 2 illustrates part of the real trace data and features we have collected from Sheepdog. In this specific example, due to the space limit, we show the following features: 1) access time, 2) object ID, 3) access length, 4) access offset, 5) operation code, 6) request ID, 7) object index, 8) object reference count, 9) zone (one

object copy per zone), and 10) target node ID. The trace data will be pre-processed for feature normalization and formatted to training datasets. These training datasets are then used by PCA (principal component analysis) module [19] to derive key features (i.e., the important, principal features) from their original features. Each feature represents one dimension describing an attribute of data accesses. Learning key features is essentially the process of dimensionality reduction. Utilizing key features and training datasets, a DBSCAN-based clustering algorithm [27] is used to group relevant objects by calculating the correlation of key features. Based on these I/O characterization results, storage systems can be tuned and optimized for better performance. We will describe the feature normalization, learning key features, clustering, and discuss the efficacy in this section.

3.1. Feature Normalization

To discover I/O characteristics, traces need to go through a feature normalization pre-processing stage to lay out a foundation for the comparison and key feature selection. The reason is that, in I/O traces, features have values in different units. For example, in Figure 2, the *access time* feature is a 64-bit integer in milliseconds, while the *access offset* feature is a 32-bit integer in bytes. Such a large range of values make it difficult to compare different features and identify their importance. To solve this issue, feature normalization is performed to reduce the range and variance of different feature values. Further, features with same or redundant values are ignored because they do not help distinguish I/O behaviors. In the current study, we also do not consider features that are not in numerical values.

There are many data normalization methods, such as min-max, z-score and decimal scaling normalization methods [28]. We use the *z-score* method for feature normalization as the trials of these methods confirm the z-score delivers the best promise to reduce the value variance across different features and remove outliers [28]. The z-score method subtracts the mean from each feature and further divides it by its standard deviation. It transforms the data set to a distribution with zero mean and unit variance. The conversion function for feature normalization of each data access is described as:

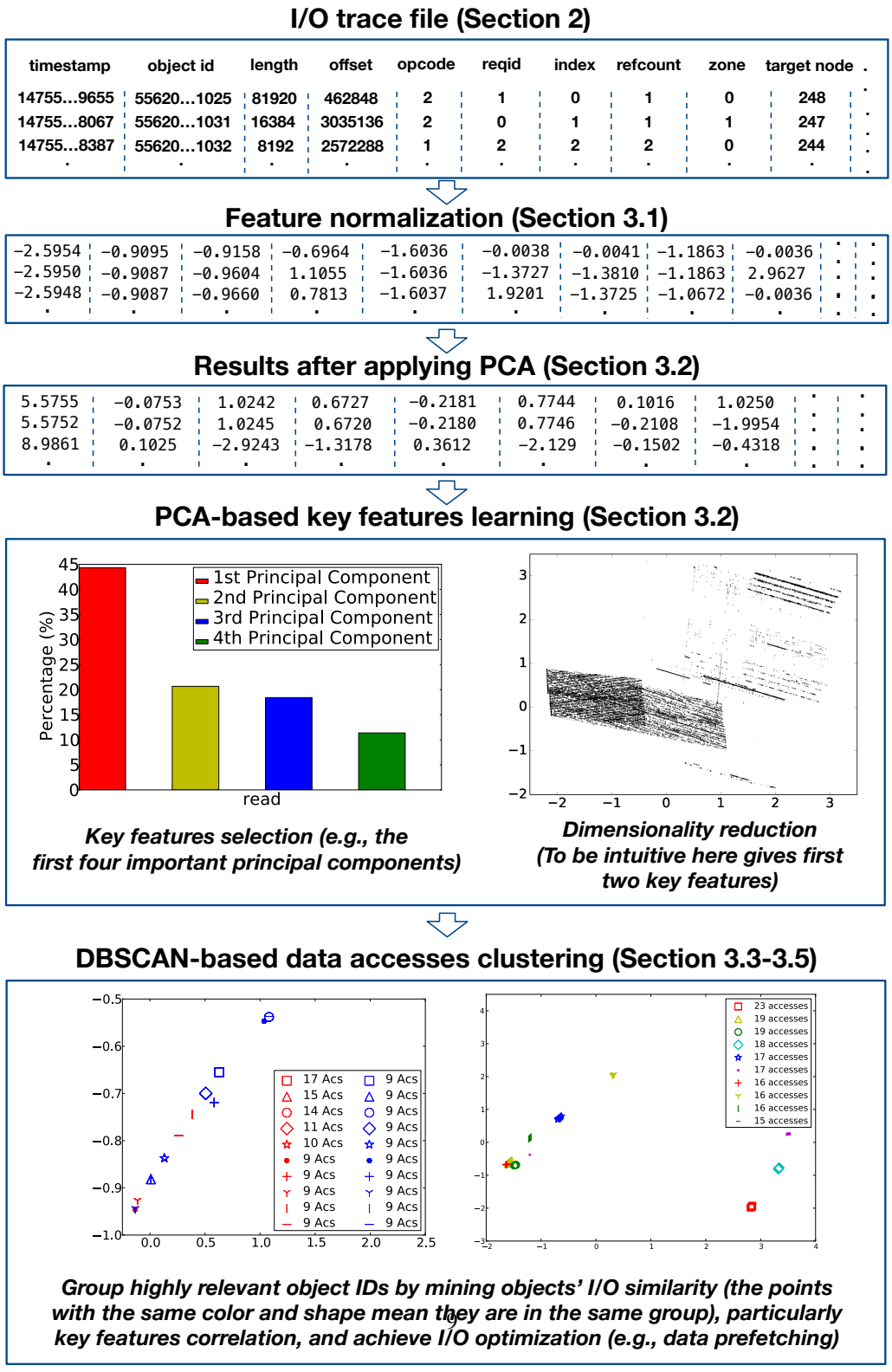


Figure 2: Overview of I/O characteristic discovery (the real trace data and features are collected on Sheepdog)

$$z = \frac{x - \mu}{\sigma} \tag{1}$$

where x is one value of the feature (e.g., 81920 in *length* feature in Figure 2), μ and σ are the average and standard deviation of all values for the feature. A sample result of feature normalization is shown in Figure 2. It can be seen that the features with large variance are transformed to the same scale, which lays out the foundation for further key feature selection.

3.2. PCA-based Key Feature Selection

After performing feature normalization, I/O access features are normalized. However, applications often exhibit complex behaviors. It is challenging to select a set of features for I/O characteristic discovery without any domain knowledge of applications [11]. To solve this problem, we introduce the concept of *principal component* to describe and identify I/O behaviors. Assume all data accesses construct a multi-dimensional space/coordinate. Each point in the coordinate represents one object access and each feature represents one dimensionality that describes an attribute of data access (e.g., the *access time* feature represents a dimension that describes when an object is accessed and the *object ID* is another dimension describing which object is accessed). The principal component analysis learns the "main direction" of data accesses with a dimensionality reduction process. This "main direction" represents the dominant data-access pattern of I/O behaviors. They keep the key set of descriptive features and reduce the noises in data accesses without requiring any expertise or domain knowledge.

More specifically, PCA (principal component analysis) method [19], an unsupervised machine learning analysis is used to learn the key features. PCA is a statistical method that captures patterns in multi-dimensional dataset by choosing a set of important dimensionality automatically, the principal components or key features, to reflect covariation among the original coordinate. The input of PCA is the set of normalized features, and the output of PCA is a new subset of features defined by the principal components, usually with less dimensionality. Each principal component has an eigenvector, which indicates the importance of this component. These eigenvectors can be

calculated from the covariance matrix in the PCA analysis. Assume the eigenvectors for n principal components are $\lambda_1, \lambda_2, ..., \lambda_n$, respectively, the eigenvector proportion of principal component i is:

$$\frac{\lambda_i}{\sum_{j=1}^{n} \lambda_j}. \tag{2}$$

We define the first k principal components as "key features", if the below proportion formula is larger than a threshold, such as 90%.

$$\frac{\sum_{j=1}^{k} \lambda_j}{\sum_{j=1}^{n} \lambda_j} \tag{3}$$

3.3. Object Clustering

With key features of data accesses, a clustering stage is performed to identify the access similarity among objects for discovering I/O characteristics. As the example in Figure 2 shows, the result of PCA is a new multi-dimensional data set, where each row/record corresponds to one original object access, and each column indicates a principal component. We use the formula (3) to select the first k principal components as the key features. Then, clustering is performed to group data accesses based on their distances calculated by the key features. As each data access has an object ID, we can consider the objects in the same group have high I/O similarity. If there are two or more data accesses for one object in the same group, we will remove duplicate objects.

We have tried three clustering algorithms, nearest neighbor (NN) [27], K-means [29] and DBSCAN [27], and selected the DBSCAN. The reasons are two-fold. First, DBSCAN is simple in terms of the algorithm complexity. It allows fast processing of large data sets with the average time complexity of $O(n \log(n))$, where n is the number of data points. On the contrast, K-means and NN are much more time consuming. K-means has the time complexity of $O(n * k * t)$, where k is number of clusters, t is number of iterative calculations. To find the k closest points, the time complexity of NN is $O(nd + kn)$, where d is the feature number of each data point.

Second, DBSCAN is a density-based clustering algorithm that is very robust and handles noisy data well. In fact, according to our observations, the output data set of

PCA stage shows irregular shapes (e.g., most data points reside close to a straight line, as seen in Figure 6 and Figure 7). In this case, K-means has low efficiency because it is used to identify a set of data points that congregate around a region in multi-dimensional space (spherical distribution) [29]. Figure 3 shows an example of K-means clustering results after PCA for real traces in Sheepdog (such a small number of data points is used for an easy illustration). The x axis and y axis represent two key features (the results have no units after the PCA stage). It can be seen that K-means algorithm will group data sets into $three$ clusters, where *cluster1* and *cluster2* cross two lines. This is counter-intuitive because data points in the same line have better similarity. Instead, DBSCAN clusters these data points along the line (data points with different colors and shapes means different clusters on the line), and the result is much more accurate. NN groups the points from two different lines into the same cluster and does not generate the accurate result.

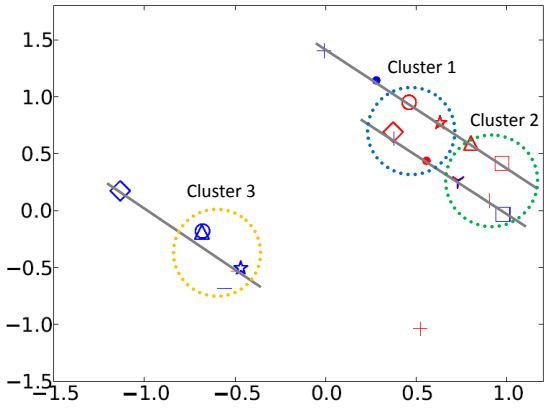


Figure 3: Object clustering with K-means

For clustering objects, DBSCAN uses a distance function to calculate the distance among data accesses to decide whether objects are in the same group. We use the Euclid space distance [27] of key features (key features correlation) to calculate the distance between two accesses. There are two parameters highly relevant with the clustering results of DBSCAN. One is a distance threshold *dis_thr* that indicates the maximum distance between two objects allowed in one group. It is actually difficult to obtain an accurate value. In this study, we present a dynamic method to adjust the distance

threshold *dis_thr*. More details will be discussed in Section 3.4. The second parameter is *min_samples*, the minimum number of objects in a group. We set the value to 2 to ensure each group at least has two data access points.

3.4. Distance Threshold Adjustment

In this section, we discuss how to tune DBSCAN to conduct an appropriate number of clusters for grouping objects. As we have described before, DBSCAN controls the clustering results through the distance threshold *dis_thr*. If this value is too small, then the average cluster size (the number of accesses in a cluster) will be very large, which cannot distinguish I/O similarity well. On the other hand, if this threshold is too large, the average cluster size will be very small, e.g., one access in the cluster, which leads to no similar objects in the system. To address this challenge, we dynamically adjust the distance threshold *dis_thr* until selecting an appropriate value. Specifically, we use the "elbow" method [30], which examines the variance of the average cluster size for different thresholds (*dis_thr*), and chooses appropriate values for both the threshold value and average cluster size.

Algorithm 1 Clustering threshold selection

procedure THRESHOLD_SELECTION($INPUT : (t_i, c_i), \ldots, (t_j, c_j)$)
  $\vec{b} = (t_j - t_i, c_j - c_i)$
  $\hat{b} = b/\mid b \mid$
  for $k = i; k++; k \leq j$ **do**
    $\vec{a} = (t_k - t_i, c_k - c_i)$
    $v = \mid a - (a \cdot \hat{b}) \cdot \hat{b} \mid$
    if $tmp < v$ **then**
      $tmp = v$
      $elbow\_point = k$
    end if
  end for
end procedure

The algorithm and pseudo-code of selecting *dis_thr* are shown in Algorithm 1. We

first calculate the average cluster size iteratively by increasing the threshold *dis_thr* from 0 with an adjustment of t_a in every step, till the average cluster size reaches a constant value. The constant value means the cluster size reachs to a stable value (e.g., equal to one), regardless how *dis_thr* changes. For each value of $thr\_adj = (t_0, t_1, \ldots, t_n)$, we use DBSCAN to calculate average cluster sizes $clu\_arr = (c_0, c_1, \ldots, c_n)$. Thus we get a curve describing the relationship between the adjustment values and the average cluster size. The points on the curve are $p = \{p_0, p_1, \ldots, p_n\}$, where $p_0 = (t_0, c_0), p_1 = (t_1, c_1), \ldots, p_n = (t_n, c_n)$. Then we remove the points whose average cluster size is one. The choice of an appropriate distance threshold *dis_thr* is to find the best *trade-off* point in the points $p' = \{p_i, p_{i+1}, \ldots, p_j\}$, where the i, j values are the indices of remaining points after removing these points whose average cluster size is one. The evaluations are shown in Section 5.1.

3.5. Re-clustering for Data Accesses

By adjusting the distance threshold *dis_thr*, we derive appropriate parameters for DBSCAN to cluster data accesses and group objects. However, in some cases, with the trained threshold *dis_thr*, there might be a few clusters with an extraordinarily large number of data points compared with others. For example, in one evaluation based on real-world traces, the sizes of almost all clusters are less than 20 expect that one larger than 300. This is because that DBSCAN is a density-based clustering algorithm and there are data points concentrating on nearly the same position in the coordinate. If we continue to reduce the trained threshold *dis_thr* at this time, it will affect the results of other clusters (i.e., each data point is regarded as a noise point). We call this extraordinarily large-size cluster "high-density cluster", which can not be well clustered through one iteration.

To solve this problem, data points in the "high-density cluster" will be identified and automatically re-clustered. A new smaller temporary threshold *dis_thr'* will be selected until appropriate cluster sizes are obtained. Figure 4 shows an illustration of high-density cluster with more than 300 points, where the x axis and y axis represent two identified key features.

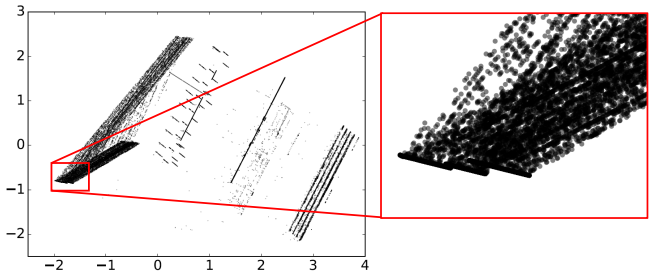


Figure 4: An illustration of high-density cluster

4. Use Case of I/O Characteristic Discovery

To validate the feasibility and to evaluate the efficacy of the newly proposed I/O characteristic discovery method, we further design and implement a *data prefetching* mechanism as a use case in this study. The prefetching mechanism is implemented on the Sheepdog storage system [4].

4.1. Data Prefetching Use Case

To speed up I/O performance, the vanilla Sheepdog provides an object cache layer (e.g., solid state disks), where objects can be cached and then flushed asynchronously into hard disks. The object cache is actually a local file system directory in the storage node for object store and independent for each storage node. The object cache can speed up reads too, but it does not work well yet due to the low cache hit ratio observed in our evaluations (varied from 2.87% to 19.83% as shown in Section 5.3). We leverage the I/O characteristic discovery method to implement a data prefetching mechanism with a hope to improve the object cache hit ratio and thus the overall performance of Sheepdog.

As show in Figure 1, I/O traces are collected on the gateway nodes via a lightweight tracing layer that periodically collects object I/O requests, where the latest accesses are used for prefetching. The *I/O characteristic discovery* (as seen in Figure 2) provides an off-line analysis process by reading the trace file and clusters objects into groups according to their similarity. Then the *object prefetching* process retrieves the clustering results from the pattern analysis and predicts future possible object accesses based on current object access. The prefetching process will send requests of predicted future

object accesses to the data retrieval thread, which is in charge of fetching objects from hard disks to the object cache. Whenever conducting a disk read, the prefetching process will first check the object cache. If the object is not in the cache yet, it then issues requests to the underlying disks. Otherwise, the prefetching process will return data to the application immediately, without fetching data from disks.

The prefetching mechanism is based on the clustering results of objects. If an object in one group is accessed, other objects in the same group are read in advance to the object cache. We use the *object ID* as a unique value to identify different objects in a group. We remove duplicate objects to make each object distinct in the group. Then an *I/O prefetch table (IOPT)* is constructed for each storage node to maintain the similarity relationship among objects. Given the key is *object ID*, the value is a candidate object list that have high I/O similarity with it (the objects in the same group). As an object may appear in multiple groups, we combine the candidate objects for it. We use binary search tree (BST) to manage the table to further minimize the search time. When reading an object, the storage node will first search it in the object cache. Upon a miss, the storage node will read the demanded object from hark disks to the cache and simultaneously prefetch objects. For each trace analysis, the IOPT will be reconstructed to reconcile the applications.

To further tune the prefetching mechanism, we consider two factors for data prefetching. One is the *object access time*. As the prefetching strategy intends to retrieve the data for future accesses, we compare the first access time of each object in the trace file with the *object ID*. Given an object x, only the objects whose access time are larger than x in the same group will be added to its candidate object list. The other factor is *object locality*. Distributed storage systems often have replicas for data placement. When prefetching an object to the object cache, we choose the copy on the local node as the first priority to reduce the latency of a remote access.

The prefetcher fetches relevant objects after analyzing the given application access patterns. It can also be applied in a setting where many applications run concurrently. It is feasible because our I/O characteristics discovery method provides a system-level I/O pattern analysis, which does not differentiate requests from concurrent applications. The traces are collected on the server-side and include the I/O requests from all

applications.

4.2. Other Use Cases

More use cases of utilizing discovered I/O characteristics could be found in practice. One example is to determine the optimal layout of data striping in parallel file systems [31]. Specifically, instead of distributing data on multiple storage nodes in a round robin fashion, file systems can take advantage of the known I/O characteristics to place relevant data strips together in the same place to minimize applications' overall data access cost. Another example is to achieve efficient data replication in distributed storage systems [24], where objects with similar I/O patterns can be identified and replicated together.

5. Evaluation

In this section, we present various experimental evaluations of the proposed I/O characterization methodology. We implemented a lightweight I/O tracing layer in Sheepdog to collect server-side traces, which include more than 20 features such as access time, object ID, length, offset, target node, etc. We also implemented the prefetching mechanism in the object cache layer of Sheepdog to evaluate the accuracy and effectiveness of identified data-access patterns.

The experiments were conducted on a local 26-node cluster, including 20 storage nodes and 6 compute nodes hosting VMs. Each storage node has dual 2.5 GHz Xeon 8-core processors, 64GB memory, a 500GB Seagate SATA HDD and a 200GB Intel SSD. The compute nodes are used for running VM clients, where each client is emulated by KVM/QEMU and configured with 2 vCPUs and 8GB RAM. We conducted the experiments using both standard file system benchmark FIO and application workload BigdataBench [32]. We used one storage node as the gateway node to collect all I/O accesses and analyze access patterns.

5.1. Distance Threshold Selection

To identify I/O characteristics, we leverage the clustering algorithm to group relevant objects according to I/O similarity. It is important to choose an appropriate dis-

tance threshold as the threshold affects the clustering results. In this section, we report the results of distance threshold selection. We launched one or six VM clients on different computer nodes to perform FIO benchmark tests.

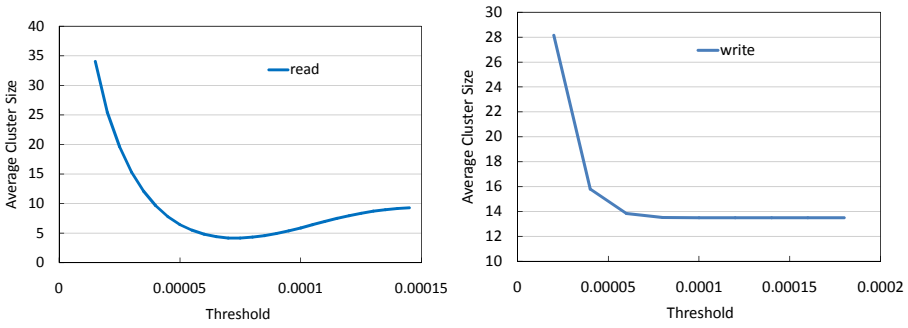
We performed four types of tests which represent different file access modes. Among them, *FIO 2Randr 2Randw 1r 1w* means tests on 6 VMs, in which two FIO for rand read, two FIO for rand write, one FIO for sequential read and one FIO for sequential write. We analyzed all the traced data (from $170K$ to $500K$ I/O requests records) and showed the training results of distance threshold adjustment in Figure 5. In this test, we set the threshold adjuster t_a as 0.00001. Though this adjustment takes multiple rounds of calculation to obtain the appropriate threshold (from 20 to 250 iterations), the total cost is actually small because each iteration time is short (average time from $1.48s$ to $6.67s$ as shown below in Table 1).

Figure 5 shows results with varying threshold values, where the x axis is threshold and y axis is average cluster size. Two observations can be made. First, the average cluster size increases with larger thresholds. It is comprehensible that more data accesses will be grouped in the same cluster if the distance threshold *dis_thr* is large. Similarly, at the beginning, the average cluster size is large because many data points are regarded as noisy data due to too small distance threshold, which in turn reduce the total number of clusters.

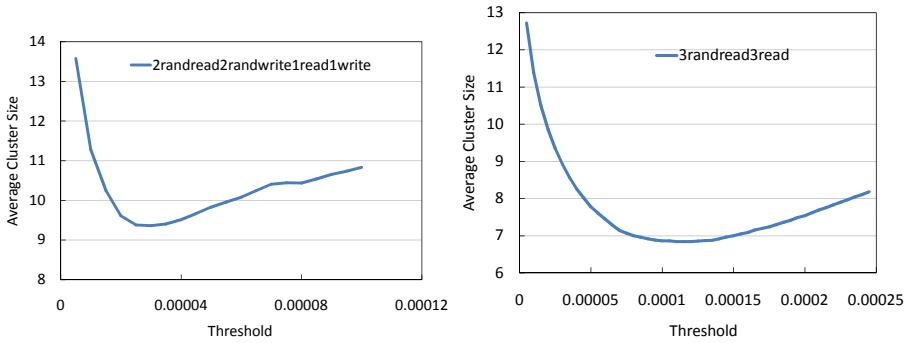
The second observation is that all tests have an "elbow position" that can be calculated in the relationship curve. Before or after the "elbow position", the average cluster size of grouping objects is not a reasonable fit for data prefetching. For example, in *FIO sequential read*, we can find the "elbow point" in threshold 0.000065. This point gives us good clustering results as the average cluster size is less than 5. For other trace results, the "elbow point" occurs in different positions with the moderate average cluster size from 7 to 13.

5.2. I/O Characteristics Analysis

In previous tests, we dynamically adjusted the threshold to select an appropriate threshold for clustering. In this section, we report the detailed results of I/O characteristics discovery. We first conducted the evaluations with FIO benchmarks on one



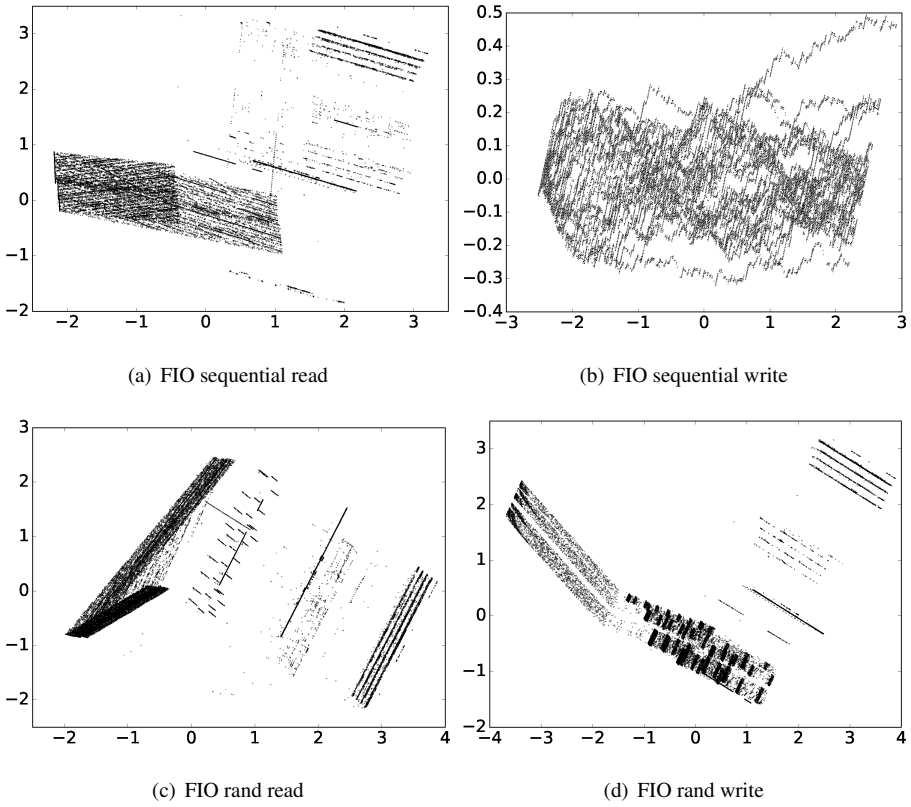
(a) FIO read (one FIO instance for sequential read) (b) FIO write (one FIO instance for sequential write)



(c) FIO 2Randr 2Randw 1r 1w (six FIO instances) (d) FIO 3Randr 3r (Three for random read and three for sequential read)

Figure 5: Relationship between the threshold values and average cluster sizes

or multiples VMs. For the test on one VM, we launched FIO with data size of $50GB$ and request size of $4MB$. For the test on multiple VMs, we launched FIO with 128 jobs, where each job accessed an independent file with $100MB$ in an asynchronous way with the request size of $4MB$.



(a) FIO sequential read

(b) FIO sequential write

(c) FIO rand read

(d) FIO rand write

Figure 6: Two key features after PCA method with FIO on 1 VM. The x axis is the 1st PC and the y axis is the 2nd PC. There are no units for key features after PCA.

Figure 6 shows the results of key features learning from I/O accesses with FIO running on 1 VM. Each point in the coordinate indicates an object access. To be intuitive, we plot $40K$ accesses based on the first principal component (1st PC) and second principal component (2nd PC) after PCA. Supposing the data accesses as points distribution in a multi-dimensional basis coordinate, the two principal components (x axis and y axis) reflect the dominant I/O behavior for I/O trace in the new 2-dimensional

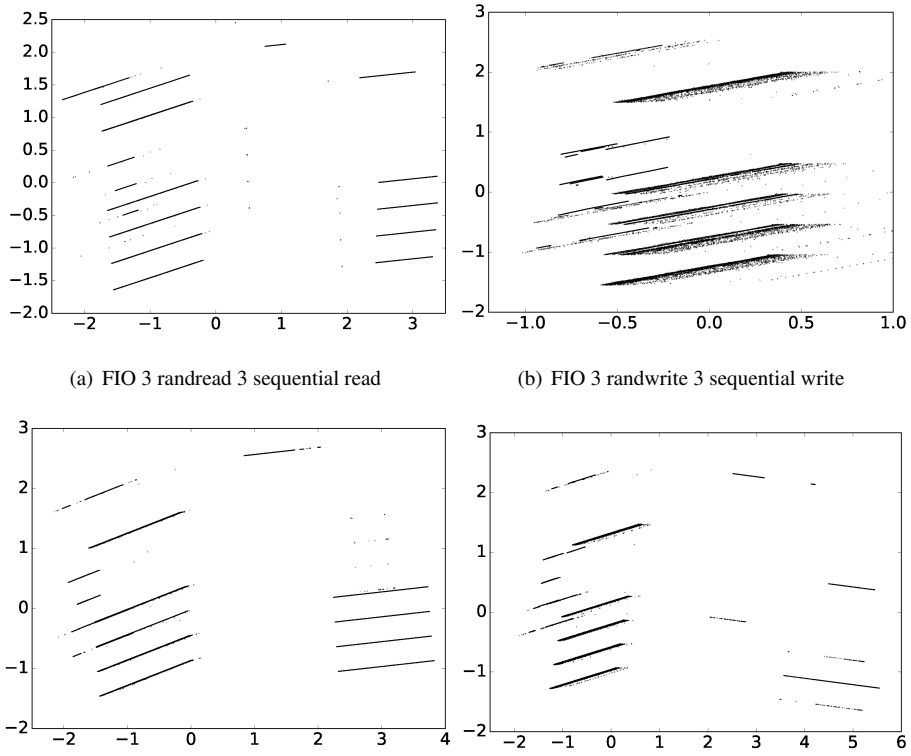
basis coordinate.

Two observations can be made from the PCA results. First, the results show that data-access patterns vary with different workloads (mean different shapes in the coordinates). But we see most data points reside close to different straight lines. All of them formed linear clusters locally and located in certain regions. The I/O similarity can be accurately identified as DBSCAN works well for this distribution. Specifically, the dark region with a large number of points means "high-density cluster", which will be re-clustered.

Second, besides *FIO sequential read*, other three benchmarks (sequential write/rand read/rand write) also have strong object similarities. One reason we infer is that the operation system in VM can also have its behaviors and affect the access patterns (e.g., the operation system call, I/O scheduling).

Similar to the test results on 1 VM, most of the data accesses construct line distribution in different regions for multiple VMs tests, as shown in Figure 7. But, unlike the patterns on 1 VM, there are few regions with "high density clusters". We can see that most of the data accesses concentrate in several parallel lines. The object similarities can also be found with DBSCAN in such scenarios.

To show the exact eigenvector proportion accounted for each principal component (which is also used for key features selection), we gave the values of first four principal components in Figure 8. It can be seen that the first principal component in the tests accounts for a large eigenvector proportion. Specifically, the 1st PC of FIO *read* accounts for up to 44.3% eigenvector proportion. Other trace can account from 27.9% to 38.2%, thus more principal components can be used until getting the major proportion. PCA does return a less dimensional data set in most cases. However, each dimensionality actually corresponds to a combination of multiple features/attributes of the original dataset, instead of representing a single, determinate selected feature. In our evaluations, although the first four principal components reach more than 90% of the variance, they do not mean only four features of the original dataset take effect. They represent the most important characteristics of I/O accesses that can be used as key features for pattern analysis. Also, Figure 8 represents only one case. In fact, if the features have large variations, more principal components will be selected as "key



(a) FIO 3 randread 3 sequential read

(b) FIO 3 randwrite 3 sequential write

(c) FIO 3 readwrite 3 randreadwrite

(d) FIO 2 randread 2 randwrite 1 sequential read 1 sequential write

Figure 7: Two key features after PCA method with FIO on 6 VMs. The x axis is the 1st PC and the y axis is the 2nd PC. There are no units for key features after PCA.

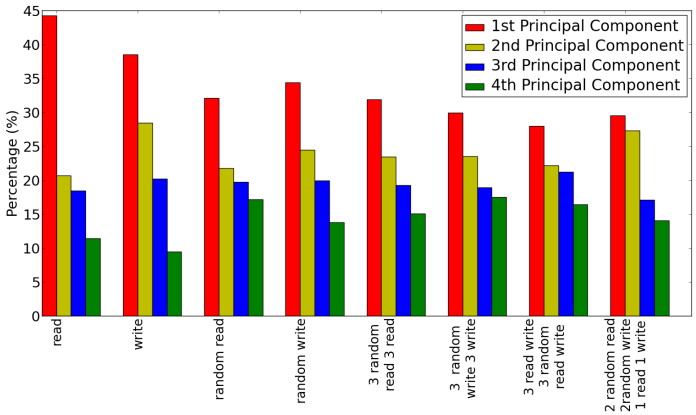


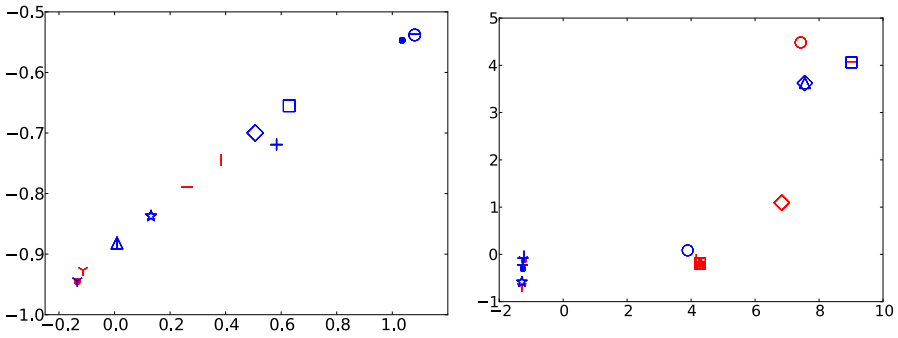
Figure 8: The proportion of variance for principal components with different FIO tests running on 1 VM and 6 VMs

Table 1: Statistics of I/O Characterization Results

Trace Type	Number of I/O Traces		Ave. Seconds for Each Iteration	Num. of Clusters	Distance Threshold
	Object Access	Whole Object Access (4MB)			
Sequential Read	171,760	3,691	1.483	32,388	0.000065
Sequential Write	386,783	113,217	6.675	29,640	0.000063
Rand read	495,465	4,535	5.361	49,217	0.005
Rand write	535,097	4,903	5.986	48,496	0.005
3Randr 3r	334,957	11,897	3.506	43,894	0.0001
2Randr 2Randw 1r 1w	228,709	6,424	1.877	20,910	0.000025

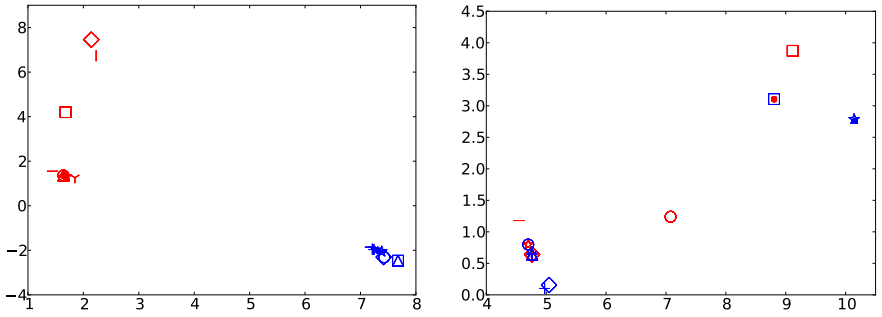
features".

Table 1 reports statistics of analysis results for different workloads. The *whole object access* means the request data size is $4MB$ (also is the object size). As the number of clusters is large, we choose the first 20 largest clusters for each test (excluding "high-density clusters", indeed the number of "high-density clusters" are less than 3 in each test), as shown in Figure 9 and Figure 10. These results are beneficial for data prefetching in two-fold. One is that we identify the objects with high similarity in I/O behaviors in the same group. The other is that the cluster sizes are appropriate for our data prefetching (The average cluster size is from 2 to 25, the maximum cluster size is less than 30).



(a) FIO sequential read

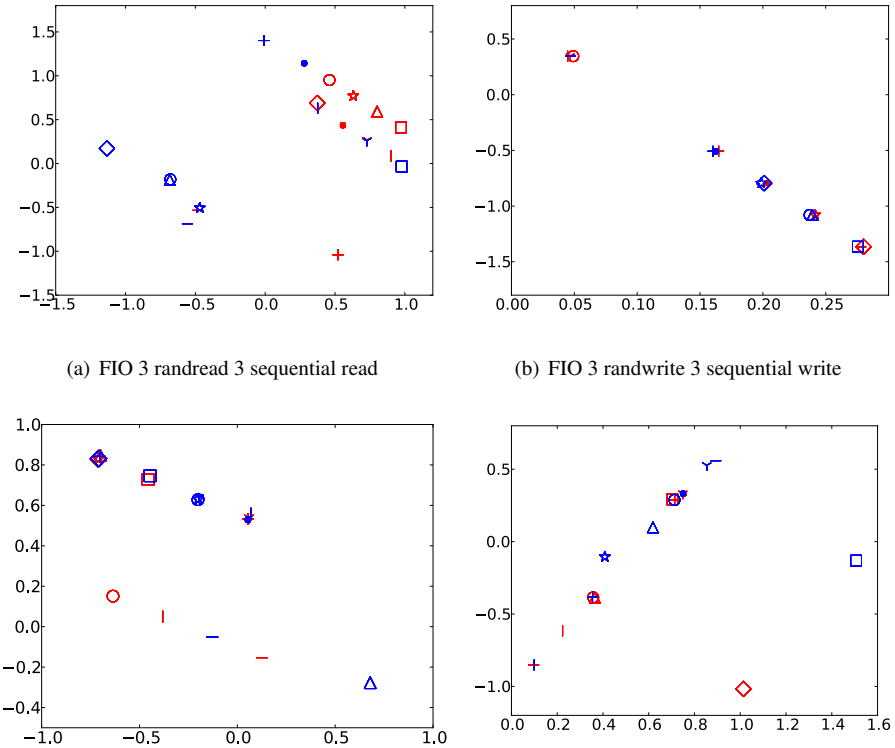
(b) FIO sequential write



(c) FIO rand read

(d) FIO rand write

Figure 9: Object clusters with 20 largest-size with FIO on 1 VM. The cluster sizes vary from 12 to 23. The points with the same color and shape mean they are in the same group. The results show strong object similarities for access patterns.



(a) FIO 3 randread 3 sequential read

(b) FIO 3 randwrite 3 sequential write

(c) FIO 3 readwrite 3 randreadwrite

(d) FIO 2 randread 2 randwrite 1 sequential read 1 sequential write operations

Figure 10: Object clusters with 20 largest-size with FIO on 6 VMs. The cluster sizes vary from 11 to 29.

Table 2: Comparison of Cache Hit Ratio (%)

Trace	FIO			Hadoop			Spark			Hive Query		
	read	Random read	3Randr 3r	Sort	Grep	Wordcount	Sort	Grep	Wordcount	Aggregation	Join	Select
Basic	8.57	6.15	7.38	3.42	5.89	2.87	13.15	16.52	12.36	6.92	8.17	9.53
Replay	10.61	8.59	9.04	4.35	6.21	3.47	15.62	19.83	14.72	7.58	9.26	11.31
Replay with prefetching	48.24	29.83	32.36	39.67	46.31	37.2	42.95	44.17	35.78	37.05	42.18	47.82

5.3. Evaluation of Prefetching Use Case

To validate the feasibility and to study the effectiveness of our methodology, we use data prefetching in Sheepdog as a use case to leverage pattern analysis results. Besides FIO, we use BigdataBench [32] to emulate real applications workloads and performed tests on three VMs. BigDataBench is a big data benchmark suite, which is widely used to emulate real-world applications and synthetic data sets. In BigDataBench, we select three application simulations, including Hadoop, Spark and Hive, for our tests. They are all popular data intensive computing applications in large-scale data centers.

We ran the application simulations to collect and analyze the I/O trace for data read operations. The vanilla Sheepdog system does not have any prefetch capabilities enabled (*basic*). To compare the performance of prefetching, we calculated the cache hit ratio by replaying the applications. We replayed the application for two times. First, we still used the default object cache layer without prefetching (*Replay*). We did it to test the ability of object cache in Sheepdog as the cache layer has been warmed with data. Second, we replayed the application with the prefetching mechanism enabled for data prefetching in the object cache (*Replay with prefetching*). During the tests, we used solid state disks as object cache storage devices with the cache capacity $10GB$. For each test, we generated $30GB$ data volume and calculated the average cache hit ratio.

Table 2 shows the comparison of cache hit ratio before and after data prefetching. We got the value of hit ratio through dividing the number of object hitting on the cache by the total number of object accesses. All tests use the default cache replacement strategy, i.e., the *random replacement*, in Sheepdog. The first two tests (*basic* and *replay*) are on the vanilla Sheeepdog without I/O prefetching, where the values vary from

2.87% to 19.83%. Although the result of *replay*) is better than that of *basic*) test (due to warmed object cache), the cache hit ratio is still low. With our prefetching strategy, the cache hit ratios increase from 29.83% to 48.24%, which are nearly improved to more than 10 times. The reason is that when an object is accessed, the objects with high similar I/O behaviors will be prefetched into the cache. Thus the system can find the future requested data in the object cache. The tests prove the effectiveness of our prefetching strategy with I/O characteristics discovery.

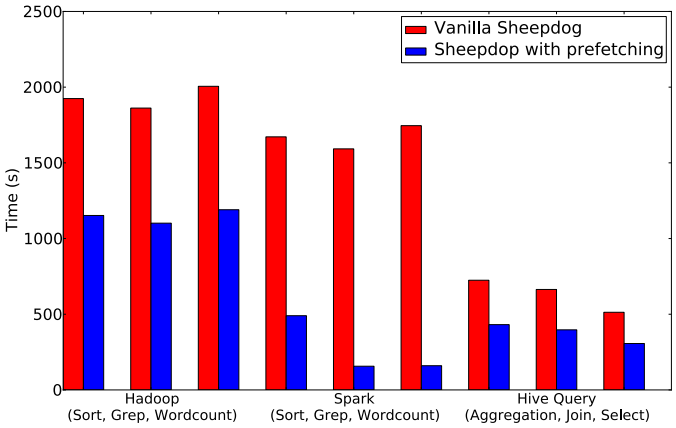


Figure 11: Overall performance improvement for data prefetching with I/O characterization

Figure 11 further describes the overall performance improvement with data prefetching for BigdataBench evaluations. With the increase of cache hit ratios, the execution time of Hadoop and Hive applications has been significantly reduced, with a percentage of up to 42%. This is because the storage system can retrieve cached objects from fast SSDs. Note that Spark achieves more performance improvement as it persists an RDD in memory [33], allowing it to be reused efficiently for the tests replay. The results show that our prefetching mechanism can significantly reduce the latency for big data applications.

6. Related Work

Numerous studies have been conducted in recent years for I/O characteristic discovery and performance optimizations. We discuss existing studies in this section and compare them with this research.

6.1. I/O profiling, tracing, and feature analysis

A number of tools have been developed to profile and trace I/O activities, such as Darshan [21], LANL-Trace [26], RIOT I/O [25], etc. Existing tools record I/O behaviors for user applications. However, most of them focus on the collection of I/O statistical information without providing effective ways to understand data-access patterns.

There is a rich set of literature on the topic of I/O features analysis and pattern discovery. These approaches mainly focus on two categories: I/O access sequence analysis and I/O semantic attribute analysis. The I/O sequence analysis is based on various parameters of data accesses, including spatial locality, temporal sequence, and repeating operations [6, 7, 8, 9, 10]. However, these analyses were performed to look for the periodicity of an application's I/O behavior based on prior workload expectations. It lacks of consideration on analyzing the I/O behaviors that have no knowledge to assume any application to possesses certain patterns.

6.2. I/O semantic attribute analysis

On the other hand, by extracting semantic attributes from file systems, semantic attribute mining approaches can analyze more complex I/O patterns and get the correlations among data accesses, such as C-Miner [13], Farmer [12], Block2Vec [34], and many others [11, 14, 15]. Although these methods look at trace for I/O characteristics discovery, they perform the pattern analysis with only one or few specific features at a time. Chen et. al. [11] proposed a multi-dimensional, statistical correlation trace analysis with K-means data clustering algorithm to identify access patterns. It can obtain comprehensive data access behavior, but require domain knowledge for selecting the set of descriptive and interpret the output results. Different from them, we introduce the *principal component* concept to automatically select key features from a vast number of access features. It reduces the bias introduced by domain knowledge or priori information of the applications. In addition, we also utilize DBSCAN, a multi-dimensional statistical data clustering algorithm to analyze I/O similarity and dynamically adjust the distance threshold for clustering. It can identify groups of highly similar I/O accesses without any assumption the number or shape of result clusters and achieve I/O

characterization.

6.3. I/O optimizations

With the analysis on data-access patterns, the storage sources can be better leveraged to boost the performance of applications. It has motivated various I/O optimizations including prefetching [10], data layout [12], and scheduling techniques [17]. Model-based algorithms, such as using neural network [7], Markov models [35], grammar-based model [9] and so on [36], have been studied and proven their efficiency for prefetching in many cases. However, existing preftching strategies mainly focus on spatial/temporal I/O behaviors or specific access features to prefetch future data and achieve performance optimization. On contrast, We address the limitations of current prediction systems for data accesses with high I/O similarity. We use PCA-based method and data clustering algorithm to analyze key feature correlations among objects from I/O behavior. With the results of pattern analysis, we can achieve an efficient data prefetching for object-based storage system and significantly reduce the I/O latency.

7. Conclusion

Many scientific and commercial applications in critical areas have become highly data intensive, which pose significant performance challenges on the storage systems. To achieve the best I/O performance, identifying and leveraging the data access pattens is a critical strategy. Numerous studies have been conducted in this space. However, most of them are either limited to specific, user-defined features for pattern analysis or heavily rely on domain knowledge about the running applications, limiting their usage.

In this paper, we have introduced a new method for I/O characteristic discovery in object-based storage systems. Different from existing approaches, this method intends to capture data-access features as many as possible to eliminate the bias for specific workloads. It utilizes the principal component analysis (PCA) to retrieve key features from traces automatically. Based on learned key features, a density-based clustering, i.e., DBSCAN, is performed to mine objects correlation and to group objects for revealing I/O characteristics. In this manner, the I/O characteristics and patterns are analyzed and discovered without any domain knowledge. We further implemented a data

prefetching mechanism on Sheepdog storage system as a use case of such I/O characteristic discovery method. This use case also serves as a mechanism to validate the feasibility of the proposed method and to evaluate its efficacy. The evaluation results confirmed that the proposed solution can successfully identify object access patterns and achieve efficient data prefetching. The buffer cache hit ratio was improved by up to 48.24% and the overall performance was improved by up to 42%. In the future, we plan to further study more use cases for I/O optimizations.

Acknowledgment

This research is supported in part by the National Science Foundation under grant CNS-1338078, CNS-1526055, IIP-1362134, CCF-1409946, and CCF-1718336. This research is also supported by Beijing Municipal Science and Technology Commission under Project No. Z191100007119002.

References

[1] M. Mesnier, G. R. Ganger, E. Riedel, Object-based storage, IEEE Communications Magazine 41 (8) (2003) 84–90.

[2] P. J. Braam, The Lustre storage architecture, White Paper (2003).

[3] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, Ceph: A scalable, high-performance distributed file system, in: Proc. of the OSDI'06, 2006.

[4] Sheepdog project (2017).
URL `https://github.com/sheepdog/`.

[5] Y. Yin, J. Li, J. He, X. Sun, R. Thakur, Pattern-direct and layout-aware replication scheme for parallel I/O systems, in: Proceedings of the IPDPS'13.

[6] Y. Yin, S. Byna, H. Song, X. H. Sun, R. Thakur, Boosting application-specific parallel I/O optimization using IOSIG, in: Proc. of the CCGrid'12.

[7] T. M. Madhyastha, D. A. Read, Learning to classify parallel input/output access patterns, IEEE Transactions on TPDS 13 (8) (2002) 802–813.

[8] T. M. Madhyastha, D. Read, Exploiting global input output access pattern classification, in: Proc. of the ACM/IEEE Conference in Supercomputing, 1997.

[9] M. Dorier, S. Ibrahim, G. Antoniu, R. Ross, Omnisc'IO: a grammar-based approach to spatial and temporal I/O patterns prediction, in: Proc. of the SC'14.

[10] J. He, J. Bent, A. Torres, G. Grider, G. Gibson, C. Maltzahn, X. Sun, I/O acceleration with pattern detection, in: Proc. of the HPDC'13, 2013, pp. 25–36.

[11] Y. Chen, K. Srinivasan, G. Goodson, R. Katz, Design implications for enterprise storage systems via multi-dimensional trace analysis, in: Proc. of SOSP'11, 2011, pp. 43–56.

[12] P. Xia, D. Feng, H. Jiang, L. Tian, F. Wang, Farmer: a novel approach to file access correlation mining and evaluation reference model for optimizing peta-scale file system performance, in: Proc. of the HPDC'08, 2008.

[13] Z. Li, Z. Chen, Y. Zhou, Mining block correlations to improve storage performance, ACM Trans. on Storage 1 (2) (2005) 213–245.

[14] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, Semantically-smart disk systems, Proc. of the FAST'03 3 (2003) 73–88.

[15] D. Dai, Y. Chen, D. Kimpe, R. Ross, Provenance-based object storage prediction scheme for scientific big data applications, in: Proc. of the IEEE International Conference on Conference on Big Data, 2014.

[16] Y. Liu, R. Gunasekaran, X. Ma, S. S. Vazhkudai, Automatic identification of application I/O signatures from noisy server-side traces, in: Proc. of the FAST'14, 2014, pp. 213–228.

[17] Y. Liu, R. Gunasekaran, X. S. Ma, S. S. Vazhkudai, Server-side log data analytics for I/O workload characterization and coordination on large shared storage systems, in: Proc. of the SC'16, 2016.

[18] L. Wang, Y. Ma, A. Y. Zomaya, R. Ranjan, D. Chen, A parallel file system with application-aware data layout policies for massive remote sensing image processing in digital earth, IEEE Transactions on TPDS.

[19] I. Jolliffe, Principal component analysis, John Wiley and Sons, Ltd, 2002.

[20] J. Zhou, D. Dai, Y. Mao, X. Chen, Y. Zhuang, Y. Chen, I/O characteristics discovery in cloud storage systems, in: Proc. of the 2018 IEEE International Conference on Cloud Computing (Cloud'18), 2018.

[21] H. Luu, B. Behzad, R. Aydt, M. Winslett, A multi-level approach for understanding I/O activity in HPC applications, in: Proc. of the Cluster'13.

[22] K. H. P. MCarns, W. Allcock, C. Bacon, S. Lang, R. Latham, R. Ross, Understanding and improving computational science storage access through continuous characterization, ACM Transactions on Storage.

[23] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, K. Riley, 24/7 characterization of petascale I/O workloads, in: Proc. of Cluster Computing and Workshops, 2009.

[24] K. Shvachko, H. Kuang, S. Radia, , R. Chansler, The hadoop distributed file system, in: Proc. of the IEEE 26th Symposium on MSST, 2010.

[25] S. A. Wright, S. D. Hammond, S. J. Pennycook, R. F. Bird, J. A. Herdman, I. Miller, A. Vadgama, A. Bhalerao, S. A. Jarvis, Parallel file system analysis through application I/O tracing, The Computer Journal.

[26] HPC open source software projects: LANL-Trace (2017).
URL `http://institute.lanl.gov/data/software/lanl-trace`.

[27] M. Ester, H. P. Kriegel, J. Sander, X. Xu, A density-based algorithm for discovering clusters in large spatial databases with noise, Kdd.

[28] L. A. Shalabi, Z. Shaaban, B. Kasasbeh, Data mining: A preprocessing engine, Journal of Computer Science 2 (9) (2006) 735–739.

[29] E. Alpaydin, Introduction to machine learning, MIT Press, Cambridge.

[30] J. D. J. Ketchen, C. L. Shook, The application of cluster analysis in strategic management research: An analysis and critique, Strategic Management Journal 17 (6) (1996) 441–458.

[31] P. H. Carns, W. B. L. III, R. Ross, R. Thakur, PVFS: A parallel file system for Linux clusters, in: Proc. of the 4th Annual Linux Showcase and Conference, 2000, pp. 391–430.

[32] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, B. Qiu, Bigdatabench: a big data benchmark suite from internet services, in: Proc. of HPCA, 2014.

[33] Apache spark documentation (2017).
URL `https://spark.apache.org/documentation.html`.

[34] D. Dai, F. S. Bao, J. Zhou, Y. Chen, Block2vec: A deep learning strategy on mining block correlations in storage systems, in: Proc. of the 45th International Conference on Parallel Processing Workshops, 2016.

[35] J. Oly, D. Reed, Markov model prediction of i/o requests for scientific applications, in: Proc. of the international conference on Supercomputing, 2002.

[36] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, J. Zelenka, Informed prefetching and caching, in: Proc. of the SOSP, 1995, pp. 79–95.