Provenance-Based Object Storage Prediction Scheme for Scientific Big Data Applications

Dong Dai[1], Yong Chen[1], Dries Kimpe[2], and Rob Ross[2]

[1]Computer Science Department, Texas Tech University
[1]dong.dai@ttu.edu, yong.chen@ttu.edu
[2]Mathematics and Computer Science Division, Argonne National Laboratory
[2]dkimpe@mcs.anl.gov, rross@mcs.anl.gov

***Abstract*—Object storage has been increasingly adopted in high-performance computing for scientific, big data applications. With object storage, applications usually use object IDs, queries, or collections to identify the data instead of using files. Since the object store changes the way data is accessed in applications, it introduces new challenges for I/O prediction, which used to work based on interfile or intrafile pattern detection. The key challenge is that the inputs of object-based applications are no longer expressed as static file names: they become much more dynamic and unstable, hidden inside application logic. Traditional prediction strategies do not work well in such conditions. In this paper, we introduce the use of provenance information, which was collected for data management in high-performance computing systems, in order to build an accurate coarse-grained (object-level) input prediction. The prediction results can be preloaded into a burst buffer to accelerate future reads. To our best knowledge, this study is the first to use provenance information in object stores to predict application inputs. Evaluation results confirm the effectiveness and accuracy of our provenance-based prediction and show that the proposed prediction system is feasible for real-work deployment.**

I. Introduction

Object storage has been considered a promising solution for next-generation, data-intensive high-performance computing platforms for scientific big data applications in both academia and industry [1], [2], [3]. It has been successfully adopted in diverse parallel file systems, such as Luster [4], [5], PVFS [6], Ceph [7], and Panasas [8]. Object storage provides applications basic storage unit as *object*, which is a logic collection of bytes on a storage device, with well-known methods for accessing and describing characteristics of the data. This object abstraction can provide the advantages of both files and blocks with better flexibility and abstraction. With the development of object storage, the way applications interact with datasets begins to change. Traditionally, to access data, applications explicitly specify the input and output files inside applications or use external parameters, and the object storage systems map these files into objects. But, more and more object storage systems are providing pure object-based APIs, which can access data based on object IDs, queries, or collections [9]. The strategy is more like interacting with database systems: applications access data based on application semantics each time. For example, in climate simulation applications, the entire dataset can be arranged as a large number of data objects, each representing the temperature, humidity, and wind velocity data on a physical spot at a given time. Under this storage model, the object collections that applications access each time may change; for instance, applications always want to read the *latest* temperature data on each location, or applications need to read all the *changed* data since the last collection.

Current storage prediction schemes can be classified into two categories: I/O sequence analysis and I/O semantic analysis. The I/O sequence analysis combines the temporal analysis and the spatial analysis of past file or block access history together for prediction. For example, Griffioen et al. introduced a way to form the sequential I/O operations within a fixed time window into a probability graph, in order to detect the correlations between file accesses [10]. C-Miner [11] and C-Miner* [12] use data-mining techniques, namely, frequent sequence mining, to discover block correlations inside files from the block access history. Markov models built on files and processes, such as MC (Markov chain) [13] and HMM (hidden Markov model) [14], have also been studied and proven efficient in many cases. On the other hand, by extracting the semantic attributes from file systems, similarity-mining approaches provide access prediction of datasets. Ellard et al. presented evidence that file attributes such as names, permission modes, and owners are highly relevant for producing properties such as access pattern, size etc [15], [16].

Both approaches heavily rely on the file structure, which may no longer exist under object storage systems. This fundamental change motivates us to design an effective prediction solution for object storage. Predicting the future input objects of an application without knowing its logic requires a detailed understanding of the application in different ways. The provenance information provides a feasible solution.

Provenance, also known as lineage, is metadata that describes the history of an object [17], [18]. In other words, the provenance describes how the object is derived and accessed. It contains detailed information about applications and datasets, which can be used to record the detailed application status. For example, the provenance of executable files can show how they were generated and started (it can record the source code files that are compiled into this program, the compiler options, and hardware/software environment variables, etc).

Although numerous research studies have addressed the design and implementation of provenance systems [17], [19], [20], [21], [22], to the best of our knowledge, there are still no object store I/O prediction systems based on provenance.

In this paper, we propose a provenance-based coarse-grained (object-level) input prediction solution for object storage in high-performance computing. The main contributions of this paper include the following:

- Analyse the possibility of input prediction within object storage and present a provenance-based solution for different scenarios.
- Design and implement a semi-supervised classification algorithm and an object collection forming algorithm to predict objects based on provenance.
- Evaluate different aspects of the proposed solution based on real-world application traces to show its effectiveness.

The rest of this paper is organized as follows. In Section II, we have a systematical analysis of the possibility of prediction in object storage systems and discuss different solutions for different cases. In Section III, we define the provenance and introduce possible strategies to collect this provenance in an HPC environment. In Section IV, we show the detailed algorithms involved in our system. In Section V, we report the experimental results based on our emulations. At last, we have the conclusion and discuss future work.

II. Prediction Analysis in Object Storage

In this section, we categorize executions into three types. For each type, we analyze its characteristics and challenges. The core idea of the provenance-based solution for each type is also briefly described.

In general, an I/O prediction system is based on the stable repetitive pattern hidden inside applications. For example, an application under development may be modified a little each time and run on the same dataset to check the correctness. Or, a frequently used scientific application may be supplied with different datasets, in order to generate new results and understandings. And statistically, applications prefer to access the "hot" datasets (locality). Based on these real-world scenarios, we divide executions into three categories, which are incrementally more difficult to predict:

1) An application runs on the same input dataset repetitively. In practice, scientists or developers commonly rerun their applications on the same dataset, changing the algorithm slightly without changing the I/O access pattern, in order to check the correctness and efficiency of different models.
2) An application runs on different input datasets each time. This situation happens when a mature scientific application or code is used. Each time, the application is run with new inputs for generating new results.
3) An application runs for the first time or changes its normal execution pattern. This situation happens rarely compared with the previous two cases.

In the first scenario, if the application repeats its I/O operations in nearly the same way, we can safely predict the future I/O operations based on its history. But, the challenge is how to determine whether the current execution will repeat some existing executions before actually running it. This turns out to answer the question: *Is there any existing execution similar to the current execution considering the I/O behavior*? To solve this problem, most existing prediction systems either require users to provide hints to describe the similarity between executions or to calculate a similarity from semantic attributes, such as file names, paths, or access mode etc [15], [16]. Both strategies have limitations. First, requiring user hints places an unnecessary burden on the developers. Second, the calculation of a similarity based on simple attributes is highly limited by user behavior: users can place irrelevant files in the same directories with a similar name, a situation that predictors cannot identify. Also similar applications can be placed in irrelevant directories with distinct names. In this research, we propose a general and accurate way to find similar executions by analyzing the provenance data. The core idea is to define the distance function among different executions based on the collected provenance information, then train different parameters of the distance function according to history I/O behaviors. We describe the algorithm in Section IV-A.

In the second scenario, repeated executions access new data objects each time. Since repeated executions share the same application, they usually follow same access patterns, which can be leveraged for prediction. Many I/O prediction studies based on file-based storage follow this idea by detecting sequential access patterns inside a file [23]. However, this strategy does not work well in object storage systems because there are no files to provide the sequential order of data streams. In our research, instead of focusing on finding internal patterns, we identify possible input collections by automatically determining the data dependence between different applications from the provenance information. This data dependence is common because many scientific applications are arranged as a complex workflow and all the applications in it are dependent. We describe the details in Section IV-B.

The last category contains those applications that have never run before or that change access patterns. Predicting their behaviors is difficult due to the lack of history information. To generate predictions for them, we inherit the work done by Griffioen et. al. [10]. The core idea is building a probability graph for all the existing data objects according to the provenance information. We can then query this graph and return the neighbors of current accessed data objects. This is not the main focus of this research and only provides a performance baseline, so we only briefly describe the procedure of building such a graph in Section IV-C.

III. Provenance Definition and Collection

The provenance definition used in this research follows the Open Provenance Model (OPM) [24], which has been standardized as a general and formal definition of provenance. OPM comprises three basic entities: the *Artifact*, *Process*, and *Agent*, and the different relationships between them. The *Artifact* means the immutable piece of state. Typical examples include files, databases, and computers etc. The *Process* represents an action or series of actions performed on or caused by artifacts and resulting in new artifacts. The *Agent* means the contextual entity acting as a catalyst of a process, enabling, facilitating, controlling, or affecting its execution. In this research, we follow the OPM model as follows:

First, we abstract five basic entities: *User*, *Job*, *Process*, *Data Object*, and *Executable File*. Each of them belongs to one of the three OPM entities: for example, *User* indicates the *Agent* entity and *Executable File* belongs to *Artifact*.

The *User* corresponds to the system users. The *Job* and *Process* indicate different executions. A job may contain multiple processes; for example, in MPI programs, each rank indicates a process, and an MPI execution (job) may contain thousands of ranks (processes). In later sections, we usually use **Execution** to describe the job or process for simplicity. The *Data Object* and *Executable File* are both users' data; but data objects represent the input/output datasets of applications, and executable files are used to start jobs by users. The executable files are also called **Applications** in later sections. A provenance collector working under our algorithm should provide provenance for these five basic entities. Users also can add their own entities to enrich it.

Based on these entities, we define five basic relations that need to be captured as provenance:

- *Run/RunBy* indicates the relation between users and jobs. A user runs a job, and a job is run by a specific user.
- *Exe/ExedBy* shows the relation between jobs and executable files. A job is always issued based on an executable file with parameters. These parameters usually are also collected as provenance.
- *Contain/BelongTo* indicates the relation between jobs and processes. A job could contain multiple processes, and each process belongs to one job.
- *Read/ReadBy* represents the relation between jobs and data objects. The job reads the data object during execution.
- *Write/WrittenBy* represents the relation between jobs and data objects. The job writes the data object during execution.

In Fig. 1, we show an example of the provenance graph based on the entities and relations defined. In this example, there are two users with three different jobs. Among them, *Job 1* and *Job 2* execute the same executable file and read the same two data objects but write into different data objects. Note that although we have defined these five basic entities and their relations, they are not the only provenance information we could cover. For example, each *Job* entity could have parameters, and each *Executable File* could have provenance information that describes how this executable file is generated.

Although there are several provenance solutions for scientific workflow systems [25], [26], a generic provenance collection tool in high-performance computing is still on the wish list. The main challenge comes from the complexity of collecting provenance from different system components in HPC systems with affordable performance penalty. In reality, we can narrow down the provenance scope and make it possible to capture the sufficient information we need.

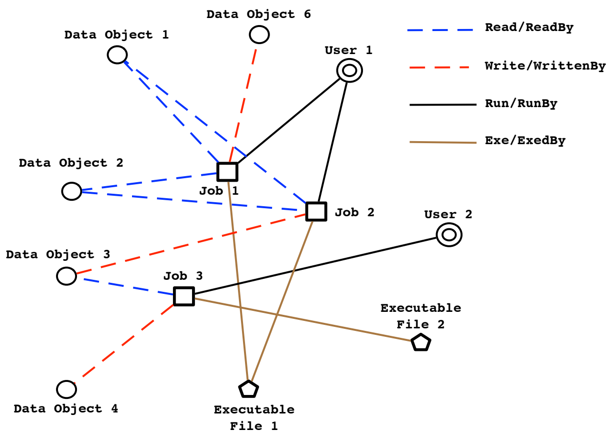

Fig. 1. Provenance graph example

One possible way of collecting the provenance about the entities and their relations described in this section is to insert the data collection stubs into different layers of the HPC software stack, for example, the job scheduler (for *Run/RunBy* relations), runtime libraries (for *Exe/ExeBy* and *Contain/BelongTo* relations), and storage libraries (for *Read/ReadBy* and *Write/WrittenBy* relations). In this research, instead of modifying the current infrastructure, we use the log traces generated while applications are running, in order to extract needed provenance information. Specifically, we use the Darshan trace [27] in this research. The reason for choosing Darshan is twofold. First, the Darshan trace is generated transparently without any modification to applications. Second, Darshan obtains a great performance by eliminating some of the detailed provenance. However, it still provides enough information to build a basic provenance graph, as Fig. 1 shows.

IV. Prediction Algorithm

Before predicting the I/O behaviors of current executions, we assume that the systems already collect history provenance and generate the provenance graph as Fig. 1 shows. So, based on our previous analysis, the entire prediction procedure can be expressed in Fig. 2: each time a new application starts, the prediction system will be fed with extra information including the user, executable file, and parameters etc. With this information, it will first try to find the similar executions in existing provenance graph which will give an accurate prediction. This case indicates the first category described in Section II. If there is not any existing similar execution, we will try to find the dependent application for the current application as the second scenario. If found, the prediction system will simply predict the possible future data object collections. If not, we will fall into the third category which contains all the remaining executions. In this scenario, the prediction system will wait for the data access requests and return the possible objects based on the probability graph of visited objects (*VisitedGraph*).

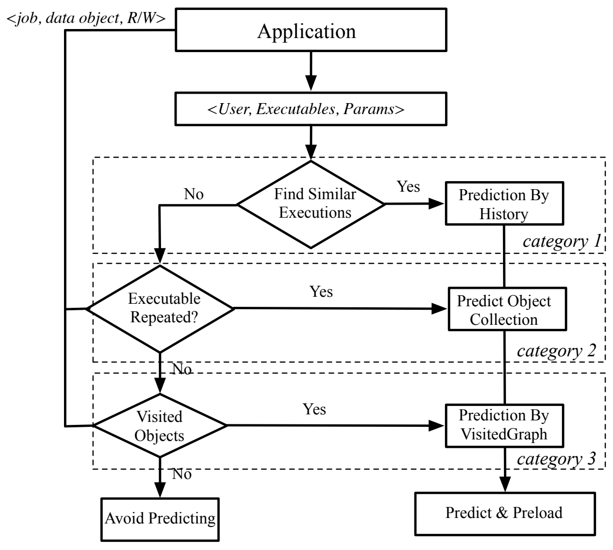

Fig. 2. Overall prediction procedure

As Fig. 2 shows, we will be able to predict the future data objects collection for most of the applications. These results are coarse-grained and useful in many situations. For example, we can preload these predicated objects into burst buffer or I/O forwarding layers to accelerate applications.

In the following three subsections, we will introduce different components and algorithms for these three different categories: the execution classification algorithm that calculates the similarity of executions, the object collection builder that creates the future object collection prediction, and the probability graph builder algorithm.

A. Find Similar Executions

In an I/O prediction system, the similarity of executions is defined as the similarity of their I/O behaviors. However, one can not know and compare the I/O behaviors among different executions before they actually finish the execution. The context-aware semantic analysis approach has been proven effective in previous works [15], [16]. The basic idea is that the similar attributes of files usually indicate the same I/O behavior. Compared with them, we argue that the provenances are more accurate than traditional semantics, simply because the provenance not only contains the semantics, such as the name, path, user, and pid etc., but also contains information including execution parameters and real data access patterns, which can be used to check how the execution was issued and behaved.

In this research, we analyze the provenances of executions to find the similar executions. The procedure includes several steps. First, we define a distance function between different executions. The distance function contains a number of *weight parameters* for each factor (described later). Then, the real data access pattern is used to train these *weight parameters*. The training strategy is based on semi-supervised classification algorithm. With these trained parameters, we can calculate the real distance between existing executions and the new execution. If the smallest distance is less than a threshold, the existing execution is considered as the similar one. We describe these steps in detail below.

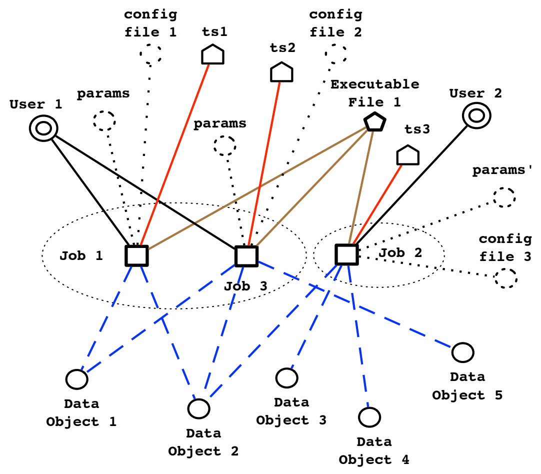

Fig. 3. Job-centric provenance graph.

1) Distance Function: Figure 3 shows an example of a job-centric provenance graph. Each job entity also can be refered to an execution. We represent the *params* and *config files* as dashed circles and edges because they are not the basic entities and relations from our provenance definition. However, this extra information can be easily collected and presented as a part of the provenance during the calculation. There are also *ts* (timestamp) entities for each of these executions. They keep an order of all executions to make sure we can distinguish any two executions, since they will have different *ts*.

We first define **provenance vector** (PV_{node_i}) for each execution. This vector contains all the relations connected with it except the *Read/ReadBy* and *Write/WrittenBy* relations.

$$PV_{n_i} = [n_j \mid n_j \in neighbors(n_i) \ \wedge \ n_j \notin Read/Write]$$

Based on the provenance vector, we define the distance function $Distance(n_i, n_j)$ as Equation 1 shows. It is calculated based on the distance of their provenance vectors.

$$Distance(n_i, n_j) = ProvDist(PV_{n_i}, PV_{n_j}); \qquad (1)$$

$$ProvDist(e_i, e_j) = \begin{cases} \sum W_r * 1 & \text{if } e_i.f \neq e_j.f \\ 0 & \text{if } e_i.f = e_j.f \end{cases}$$

The distance of two provenance vectors ($ProvDist(e_i, e_j)$) contains the differences of all provenance fields (each field means a different relation). Each difference in the fields contributes a '1' in the distance score. For example, if the *User*s of two executions are different, then we get a "1" to denote this difference. A weight (W_r) is introduced to represent the importance of each provenance relation in calculating the distance of two executions. W_r is also the training target of our semi-supervised algorithm. Each executable file should have its own W_r, since different applications may be sensitive to different provenance fields. For example, for some applications, a different user can change its I/O behavior; but for others, only the parameters will change the I/O behaviors.

2) Classification Algorithm: With the distance function, now we can measure the distance between any two executions to find their similarity. However, the large number of executions makes this approach not practical. In fact, as an optimization, we only need to calculate distance between executions from the same *executable files (application)* as they are the ones that are expected to have the simliar I/O behaviors. The target of the classification algorithm becomes to classify all executions of the same application into different groups, so that all the executions belonging to the same group share similar I/O behavior. For example, as Fig. 3 shows, *Job 1* and *Job 3* should belong to the same group, and *Job 2* should have its own group.

Two restrictions make this challenging. First, the group number is not fixed: the algorithm should add or remove groups accordingly. Second, the weights in distance function are not fixed: they should be adjusted by the real data access pattern. These challenges lead us to train the weight parameters in a semi-supervised way.

The training algorithm works as Alg. 1 shows. First, we define io_thr as the maximal I/O distance. If two executions have the same I/O behavior, then their I/O distance should be smaller than io_thr. Second, we define (thr) as the maximal provenance distance. If executions are classified into the same *group*, they should have a smaller provenance distance than thr. Moreover, we defined two learn factors (positive learn factor and negative learn factor) as the minimal adjustment on the weights each time.

During training, the *Train* function will first call *min_io_diff(groups, Exe)* to find the nearest group and the minimal distance based on the real I/O behavior. If the minimal I/O distance is less than io_thr, then we know that this new execution should belong to the same group (g). Based on existing weights ($W_r[app]$), we can calculate the provenance distance using *ProvDist()* function. If the provenance distance is larger than thr, we need to decrease the parameters weights to reduce the distance until the distance is smaller than thr.

If the actual I/O distance is larger than io_thr, then this execution has a distinct I/O behavior and should be placed into a new group. After creating a new group, we need to adjust the $W_r[app]$ by increasing the weights to make sure all other groups are far from current executions. The adjusting is simple: in adjust_increase($W_r[app]$) we *add* a small constant to the weight of each different relations, and in adjust_decrease($W_r[app]$) we *reduce* a small constant to the weight of each different relations.

Before predicting the I/O behaviors, we use the existing provenance to train all the weights for each application (executable file) offline. So, when a new execution comes and needs to predict, we will check whether it is based on an existing application. If yes, find out the group that has the smallest provenance distance with this execution. If there is

Algorithm 1 Semi-supervised Classification Algorithm

```
 1: procedure TRAIN($Exe$)
 2:     app = get_app_of_execution(Exe)
 3:     groups = get_groups_of_app(app);
 4:     (diff, g) = min_io_diff(groups, Exe);
 5:     if diff $\leq io\_thr$ then
 6:         while ProvDist(g, Exe) $\geq thr$ do
 7:             adjust_decrease($W_r[app]$);
 8:         end while
 9:     else
10:         gn = create_new_group(groups, Exe);
11:         for all $group$ except gn do
12:             while ProvDist(group, Exe) $< thr$ do
13:                 adjust_increase($W_r[app]$);
14:             end while
15:         end for
16:     end if
17: end procedure
```

any, the read behaviors of this group are returned as the prediction. If no such group exists, we move to next phase, which is trying to build the object collections based on the dependency between applications.

3) Complexity Control: Since classification algorithm needs to run on all the history logs, efficiency is critical. Generally, the time complexity of the classification algorithm is $O(K * n * t_{dist})$, where K equals the number of groups and n denotes the execution numbers. The group number (K) indicates the number of distinct execution behaviors from the same executable files (applications). It would not be large in a real-world environment. Besides, the executions number (n) can be easily reduced by choosing one execution from an execution group to represent the group instead of calculating distances for all the existing executions. The main complexity comes from t_{dist}, which represents the calculation time of distance function on two executions, determined by the number of provenance records involved in this calculation.

In general, the records collected from provenance collection tools can be too large to be calculated. For example, the *am-utils* PASS trace, which was collected using the PASSv2 research kernel [17] in real applications [28], contains around 40k provenance records for processes and thousands of provenance records for files. All these records are collected while building one middle-sized, open-source software.

Clearly, we need to reduce the provenance records number to control the complexity. Because these provenance records are used only to identify the possible I/O behavior difference, most of them are actually useless. In the proposed prediction definition, we eliminate the provenance records about system-level tools, libraries, and their parents. Doing so significantly reduces the provenance records number. We test the computation time of the classification algorithm running on different record sizes using the *am-utils* PASS trace, as Table I shows. We observe that the compression significantly reduces the execution time of the classification algorithm and makes it possible in real-world deployment.

TABLE I
TIME SPENT OF CLASSIFICATION ALGORITHM

Compress Levels (Provenance Records)	**Execution Time**
Original Provenance (≈40K)	1150 ms
Remove System Provenance (≈10K)	277 ms
Remove Duplication (≈5K)	107 ms
Compress Files (≈100)	7 ms

B. Build Objects Collection

As Fig. 2 shows, for those executions that repeat existing executable files (application) but can not be classified into any existing groups (*category 2*), we will predict their possible future data objects collection instead of an accurate data access sequence. In practice, if we run the same applications with different inputs, it is likely that we are processing some data newly generated. This simple fact leads to a possible prediction: other applications running in the system generated the data that the current run may access. This situation happens often, especially in a workflow system. Each time a new workflow is started on an initial dataset, each job except the first one, will read data generated from its ancestor. In this research, we try to identify these dependent jobs inside workflow to predict input data objects for an execution.

We call one application app_r is the *pair application* of another application app_w when app_r reads most of the outputs of app_w. Note that the application discussed here is the *Executable File* in the provenance definition. Knowing two applications are paired, the new execution of app_r has a high possibility of reading the data generated by the most recent execution of app_w. Then, we can build the future object collections of the read application based on the outputs of the other application.

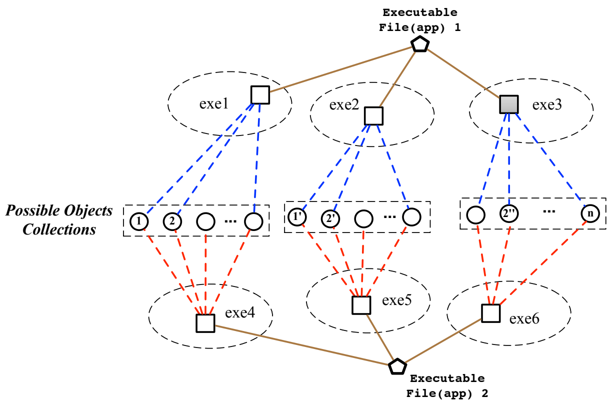

Fig. 4. Build an automatic object collection

Figure 4 demonstrates an example of paired applications (we use *Executable File* to denote different applications in this figure). The ancestor and descendant relationships between the executions and data objects were collected from provenance. From this example, one can easily see that app_1 and app_2 are paired as they share the same objects except that app_1 reads them and app_2 creates/writes them. In such pattern, although each time app_1 reads a new dataset, we can still predict an accurate objects collection.

The algorithm works on historical logs to identify this pattern. For each application, we first obtain both its *output* objects list and its *input* objects list. Each time the application executes to read or write data objects, we will added those I/O behaviors into its *input/output* objects list. After that, if the execution currently is reading a data object (DO_i), we further find out the applications that have written into the same data object by traveling back through the *WrittenBy* relation of DO_i. Then, we are able to calculate the overlap ratio between them as follows:

$$OverlapRatio(app_r, app_w) = \frac{|inputs(app_r) \cap outputs(app_w)|}{|outputs(app_w)|}$$

The overlap ratio denotes the percentage of data objects that are read by app_r from all the outputs of app_w. A higher ratio shows a higher possibility that app_r is dependent on app_w and provides a more accurate prediction for possible object collection. We will discuss this in the later evaluation section.

C. Build VisitedGraph

For the executions that are left, we predict their behaviors based on the probability graph generated from the existing provenance information. As Fig. 3 has shown, each execution will access a number of data objects. We consider that the data objects visited by the same execution are related and add an edge between these two data objects to build the *VisitedGraph*.

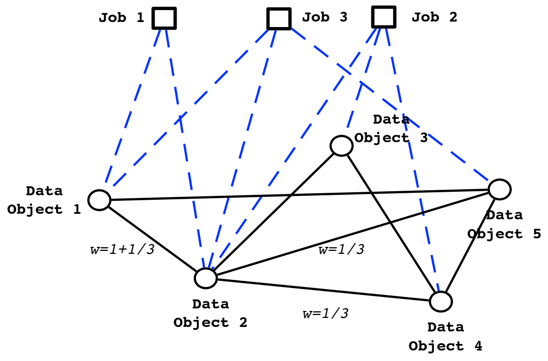

Fig. 5. An example of visited probability graph

Figure 5 shows an example of the visited probability graph generated from the provenance. The blue dashed lines represent the read operations on the data objects from different jobs. If two data objects were read by the same job (also refered as execution), we will connect them with a solid line. For example, there is a line between *Data Object 1*(DO_1) and *Data Object 2*(DO_2) because they are both visited by *Job 1* and *Job 2*. Each edge also has a weight, which denotes the probability of visiting these two data objects in one execution. Assume an execution visits k data objects; then there will be $k(k-1)/2$ edges between any two data objects. We consider that each execution provides a constant weight 1, so each edge created from this execution will share the same amount of weight: $\frac{2}{k(k-1)}$. If an edge already exists, the new weight will still be added on the old value. In Fig. 5, the edge between DO_1 and DO_2 has a weight $4/3$, and the edge between DO_2 and DO_3 has a weight $1/3$.

The component of *prediction by VisitGraph* is straightforwards. It is based on this probability graph: whenever a data object access request arrives, we check whether it has edges belongs to the *visitedGraph*. If yes, we return all its neighbors whose weights are larger than a predefined threshold.

V. Evaluation

In this section, we present the evaluation results of the proposed object store prediction system from two aspects. First, we present the results of several microbenchmarks tested for different components of our prediction system to evaluate the accuracy and effectiveness. Second, we present the overall accuracy and the results of the prediction on real data traces.

A. Methodology

Evaluating of proposed system is challenging. Although the object storage and provenance system have been widely studied, there are still no production-level storage systems with both object store and provenance support in HPC environment. In this research, we evaluate the system, especially the core algorithms, based on the emulations from several open data traces described below.

1) Darshan Data Trace: As we have discussed in the provenance collection section, Darshan [27] trace already collects many useful provenance information. In the evaluation, we used the Darshan trace data collected from a current leadership supercomputer : Intrepid [29] to verify our proposed algorithms. Specifically, the trace data includes all the real applications running in the October of 2013 [30] in Intrepid. During that period, around 200 different applications were running on the Intrepid supercomputer, and these applications total executed more than 13K times. The Darshan trace, however, contains only file statistics, since Darshan is developed for file-based storage systems. We conducted preprocessing on the Darshan trace and map each file as an object in our evaluations.

2) Workflow Provenance Data Trace: In addition to evaluate with the Darshan trace, we also used the Gigabyte Synthetic Database (GSD) as another evaluation trace. It is a noisy data collection generated using the Workflow Emulator Tool (WORKEM [31]) with a number of scientific workflows. These workflow provenance provide a good scenario to check the correctness of proposed object collection building algorithm. To unify the way of collecting provenance, we translate the GSD data trace (XML format) into the Darshan formation.

The GSD includes six workflows: LEAD NAM, SCOOP ADCIRC, NCFS, Gene2Life, Animation, and MotifNetwork. Table II shows the detailed workflow structure. These workflows are pseudo-realistic in the sense that they are modeled after real-life workflows using the WORKEMs task state model. LEAD NAM, SCOOP, and NCFS are weather and ocean-modeling workflows; Gene2Life and MOTIF are bioinformatics and biomedical workflows; and the Animation workflow carries out computer animation rendering. From this data trace, we were able to evaluate the accuracy and performance of object collection builder.

TABLE II
Overview of Workflow Structure

Name	**Number of Nodes**	**Number of Edges**
LEAD NAM	6	11
NCFS	7	19
SCOOP	6	10
Gene2Life	8	15
Animation	22	42
Motif	138	275

B. Evaluations of Classification

In our first set of tests, we applied the classification algorithm to Darshan data traces in order to evaluate the correctness of the classification algorithm. We first trained the parameters based on the Darshan trace data collected in September 2013 and then predicted all the executions in October 2013.

TABLE III
Classification Evaluation on Darshan Trace

Total Exes	**Repeated**	**Accuracy**	**Coverage**
13491	12967	12930 (99.7%)	96%
	Repeated(Less)	**Accuracy**	**Coverage**
1079	216	179 (82.8%)	20%

In this evaluation, we consider both the accuracy and coverage of the predictions. The accuracy indicates the percentage of executions whose I/O behavior followed the predictions. The coverage shows how many executions we can make predictions from our algorithm. As Table III shows, for all the executions, the accuracy of our prediction is around 99%. The result is impressive considering that the predictions were for almost 96% (12967/13491) of all the executions. The reason for such a high accuracy is that one executable file was repeated more than 12K times in that month, all having the similar I/O behavior. If we eliminated these executions, we still achieved a 82.8% accuracy and 20% coverage, as the second row shows. In this experiment, the thr was set as 0.3, and the positive and negative learning speed was 0.25 and 0.05 respectively. We consider that two executions have the same I/O behaviors if the data objects accessed have less than 20% difference.

We note that Darshan currently collects only a small number of relations (user, start time, nprocs, and several metadata fields), and hence the results should be considered as preliminary. However, even with the limited provenance, current results still show an impressive accuracy and coverage of the proposed object store prediction scheme.

C. Object Collection Builder Evaluations

In the next set of tests, we evaluated the workflow-level provenance data trace called Gigabyte Synthetic Database (GSD) described before. This trace contains repetitive executions of six well-known scientific workflows. Each of these workflows contains a fixed dependence among all the application nodes. Our target is to find all the dependent application nodes inside a workflow as paired executions. Figure 6 shows the evaluation results. The x-axis shows different workflows, and the y-axis shows the ratio of paired executions we identified. The result shows that for all these workflows, the object collection builder is able to reveal accurate paired executions.

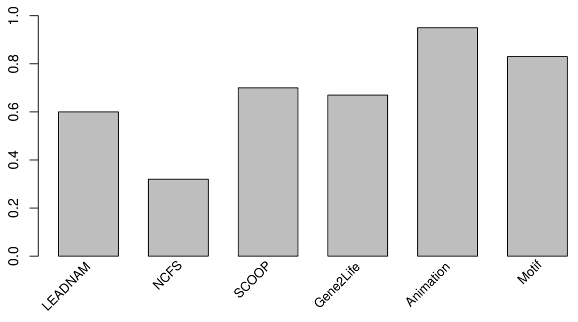

Fig. 6. Ratio of found paired executions in six workflow applications. (Overlapping Threshold ($thr_{overlap}$) is set to 0.6 in this case.)

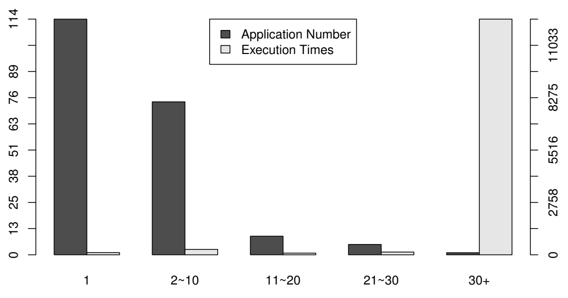

Fig. 7. Repetitive executions distribution

In addition to testing on the GSD workflow provenance, we also evaluated the algorithm on Darshan data trace, which does not contain explicit workflow dependence between different applications. Suprisingly, we find that most of the executions in the Darshan trace are dependent on themselves by repeatedly reading their own outputs. Figure 7 shows the distribution of these repetitive applications: x-axis denotes the range of repetitive times and the left y-axis bar shows the number of applications (executable files) falling into each category. For example, around 70 applications are repeated 2 to 10 times to

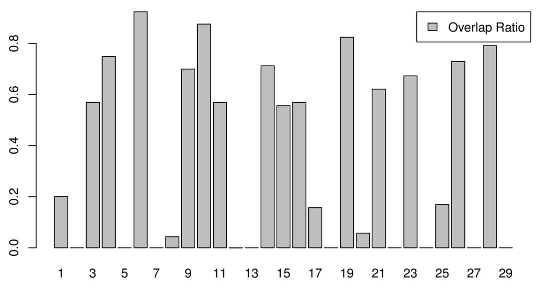

Fig. 8. Paired Executions in Darshan trace data and their overlap ratios

itself. The right y-axis bar indicates the number of executions (not applications) falling into each category. For example, most executions (95.2%) are actually repeating more than 20 times, and executions that happen only once contribute merely 0.1%.

In Fig. 8, we further show 29 typical repetitive applications that read their own previous outputs. For each application, we calculated the overlap ratio defined in Section IV-B. As we can observe, most of these executions have a high ratio, and some of them are even close to 1. The $thr_{overlap}$ is an important parameter in the object collection builder. In Fig. 9, we show how different thresholds (from 0.1 to 1.0) affect the *paired executions* numbers in the Darshan data trace. Based on this chart, we chose 0.6 as the $thr_{overlap}$ in our prediction algorithm, which means that at least 60% of one execution's outputs will be read by its *paired execution*.

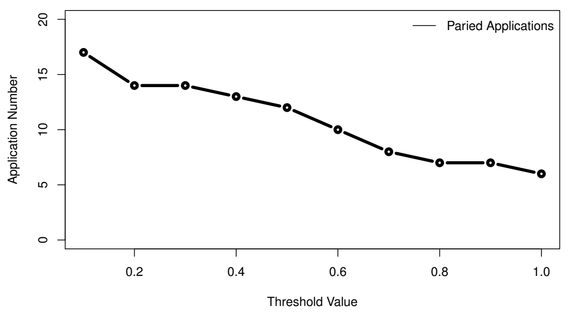

Fig. 9. Paired execution number with different threshold values

D. Overall Prediction Performance

To evaluate the overall prediction performance, we deploy the algorithms on all the executions of the Darshan trace online. All these logs were ordered, and objects accesses were predicted for each read operation. Fig. 10 shows the prediction accuracy for three typical applications in the Darshan trace. The x-axis denotes the whole lifetime of an execution, and we normalized them based on the shortest one. In Fig. 10, *apptype 1* denotes similar executions that run on changing datasets, and they were changed by other executions. As the chart shows, our prediction system achieve more and more accurate predictions as time goes by. The *apptype 2* shows applications that read new inputs. Being different from apptype 1, these new inputs were not generated by other executions. We can observe that the accuracy was initially high, but was gradually reduced. The reason is that there is no enough information to predict these completely new inputs. The *apptype 3* is the most repetitive application. As we can observe, it achieved a nearly constant prediction accuracy (80%).

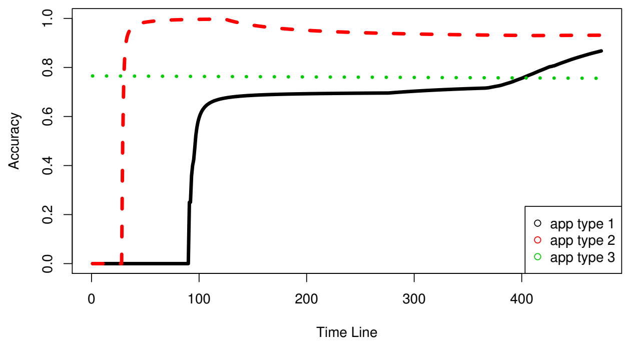

Fig. 10. Prediction accuracy of different applications.

VI. Conclusion

In this paper, we have proposed a provenance-based object storage access prediction scheme for scientific big data applications in high-performance computing. It is the first time provenance information of applications and data objects has been used to make predictions in an object store. The foundation of this research work relies on reasonable assumptions and simplifications that we have made on object storage systems. During designing and prototyping the proposed prediction system, we introduced two new algorithms based on provenance analysis to classify the executions into different groups and build objects collection for an execution. Due to the limitation of existing object storage systems, we provided emulation-based evaluations based on different sets of data traces to show the accuracy and effectiveness of our algorithms. The preliminary results show that the new prediction system is accurate and efficient. We believe that this general object I/O predictions could be widely used in prefetching, prestaging, and data reorganization for scientific big data applications in high-performance computing.

VII. Acknowledgments

This research is sponsored in part by the National Science Foundation under grant CNS-1338078 and CNS-1162488. This work was also supported by the U.S. Department of Energy, Office of Science, under Contract No. DE-AC02-06CH11357.

References

[1] B. Welch and G. A. Gibson, "Managing scalability in object storage systems for hpc linux clusters," in *MSST*. Citeseer, pp. 433–445.
[2] M. Mesnier, G. R. Ganger, and E. Riedel, "Object-based storage," *Communications Magazine, IEEE*, vol. 41, no. 8, pp. 84–90, 2003.
[3] M. Factor, K. Meth, D. Naor, O. Rodeh, and J. Satran, "Object storage: The future building block for storage systems," in *Local to Global Data Interoperability-Challenges and Technologies, 2005*. IEEE, pp. 119–123.
[4] P. Schwan, "Lustre: Building a file system for 1000-node clusters," in *Proceedings of the 2003 Linux Symposium*, vol. 2003.
[5] P. J. Braam, "The lustre storage architecture," 2004.
[6] R. B. Ross and R. Thakur, "Pvfs: A parallel file system for linux clusters," in *in Proceedings of the 4th Annual Linux Showcase and Conference*, pp. 391–430.
[7] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006, pp. 307–320.
[8] B. Welch, M. Unangst, Z. Abbasi, G. A. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou, "Scalable performance of the panasas parallel file system," in *FAST*, vol. 8, pp. 1–17.
[9] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross, "Understanding and improving computational science storage access through continuous characterization," *ACM Transactions on Storage (TOS)*, vol. 7, no. 3, p. 8, 2011.
[10] J. Griffioen and R. Appleton, "Reducing file system latency using a predictive approach," in *USENIX Summer*, pp. 197–207.
[11] Z. Li, Z. Chen, S. M. Srinivasan, and Y. Zhou, "C-miner: Mining block correlations in storage systems," in *FAST*, pp. 173–186.
[12] Z. Li, Z. Chen, and Y. Zhou, "Mining block correlations to improve storage performance," *ACM Transactions on Storage (TOS)*, vol. 1, no. 2, pp. 213–245, 2005.
[13] D. Joseph and D. Grunwald, "Prefetching using markov predictors," in *ACM SIGARCH Computer Architecture News*, vol. 25. ACM, pp. 252–263.
[14] J. Oly and D. A. Reed, "Markov model prediction of i/o requests for scientific applications," in *Proceedings of the 16th international conference on Supercomputing*. ACM, pp. 147–155.
[15] D. Ellard, M. Mesnier, E. Thereska, G. R. Ganger, and M. Seltzer, "Attribute-based prediction of file properties," *Harvard Computer Science Group Technical Report TR-14-03*, 2003.
[16] P. Xia, D. Feng, H. Jiang, L. Tian, and F. Wang, "Farmer: a novel approach to file access correlation mining and evaluation reference model for optimizing peta-scale file system performance," pp. 185–196, 2008.
[17] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. I. Seltzer, "Provenance-aware storage systems," in *USENIX Annual Technical Conference, General Track*, pp. 43–56.
[18] J. Freire, D. Koop, E. Santos, and C. T. Silva, "Provenance for computational tasks: A survey," *Computing in Science & Engineering*, vol. 10, no. 3, pp. 11–21, 2008.
[19] K.-K. Muniswamy-Reddy and M. Seltzer, "Provenance as first class cloud data," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 4, pp. 11–16, 2010.
[20] D. Zhao, C. Shou, T. Malik, and I. Raicu, "Distributed data provenance for large-scale data-intensive computing," in *IEEE International Conference on Cluster Computing, IEEE CLUSTER*, vol. 13.
[21] K.-K. Muniswamy-Reddy, U. Braun, D. A. Holland, P. Macko, D. Maclean, D. Margo, M. Seltzer, and R. Smogor, "Layering in provenance systems," in *Proceedings of the 2009 USENIX Annual Technical Conference*.
[22] A. Gehani and D. Tariq, "Spade: Support for provenance auditing in distributed environments," in *Proceedings of the 13th International Middleware Conference*. Springer-Verlag New York, Inc., pp. 101–120.
[23] J. He, J. Bent, A. Torres, G. Grider, G. Gibson, C. Maltzahn, and X.-H. Sun, "I/o acceleration with pattern detection," in *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*. ACM, pp. 25–36.
[24] L. Moreau, B. Clifford, J. Freire, J. Futrelle, Y. Gil, P. Groth, N. Kwasnikowska, S. Miles, P. Missier, J. Myers *et al.*, "The open provenance model core specification (v1. 1)," *Future Generation Computer Systems*, vol. 27, no. 6, pp. 743–756, 2011.

[25] B. Clifford, I. Foster, J.-S. Voeckler, M. Wilde, and Y. Zhao, "Tracking provenance in a virtual data grid," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 5, pp. 565–575, 2008.
[26] Y. L. Simmhan, B. Plale, and D. Gannon, "Karma2: Provenance management for data-driven workflows," *International Journal of Web Services Research (IJWSR)*, vol. 5, no. 2, pp. 1–22, 2008.
[27] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley, "24/7 characterization of petascale i/o workloads," in *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*. IEEE, 2009, pp. 1–10.
[28] PASSTrace, "http://www.eecs.harvard.edu/syrah/pass/traces/."
[29] Intrepid, "http://www.top500.org/system/176322."
[30] "http://www.mcs.anl.gov/research/projects/darshan/data/."
[31] L. Ramakrishnan and D. Gannon, "A survey of distributed workflow characteristics and resource requirements," *Indiana University*, 2008.