File Type Classification for Adaptive Object File System

Phond Phunchongharn
Department of Computer Engineering
King Mongkut's University
of Technology Thonburi
Bangkok, Thailand
p.phond@gmail.com

Supart Pornnapa
Throughwave (Thailand) Co.,Ltd.
Bangkok, Thailand
part@throughwave.com

Tiranee Achalakul
Department of Computer Engineering
King Mongkut's University
of Technology Thonburi
Bangkok, Thailand
tiranee@cpe.kmutt.ac.th

***Abstract*-This paper gives an overview of a novel storage management concept, called, Adaptive Object File System (AOFS). The design of the file type classification module in AOFS is emphasized. The design attempts to increase the efficiency through the dynamic tuning technique, which automatically classifies files using attributes and access pattern. The file classification, thus, allows files to be stored in the most efficient way. The key idea is to utilize the file properties, such as access pattern, owner, size, and permissions to select adaptive file system policies (e.g. disk allocation, redundancy, and caching strategies). Moreover, a metadata store is maintained to provide the best possible dynamic tuning strategy for any given operating period. The static classification initial design is done based on decision tree, while the dynamic classification adapts the Hidden Markov Model for prediction. The main goal of the AOFS design is to enhance system performance, storage efficiency and flexibility.**

I. Introduction

The processor performance has been increased about 50% every year in the past decade [1], while the disk access time only has an 8% improvement [2]. Such a large performance gap makes the overall system efficiency dropped. Consequently, there is a real need for research in the area of storage system.

Currently, the well-know technique used in storage optimization is striping [1]. RAID and Parallel File System both adapt the technique, where blocks of data are striped across multiple disks allowing multiple blocks to be accessed in parallel [3]. To decrease the latency time, the staging predictive caching can also be used to pre-fetch data [4]. In Database File System (DBFS), metadata of file properties is kept in order to reduce the file searching time [5].

Unfortunately, most commercial file systems that adapt these techniques do not take file access pattern into account. All types of files are thus stored indistinctively as data blocks over storage devices. Moreover, striping technique cannot extend existing data blocks over new devices/volumes. The total system access efficiency is thus not optimized. Different storage capability parameters are also not utilized to improve storage performance.

In this research, we propose an alternative approach, called, Adaptive Object File System (AOFS) as shown in Figure 1. AOFS is designed to operate in a heterogeneous environment with various operating systems and storage devices. AOFS creates an abstract layer to separate the actual physical storage from the user land application. The

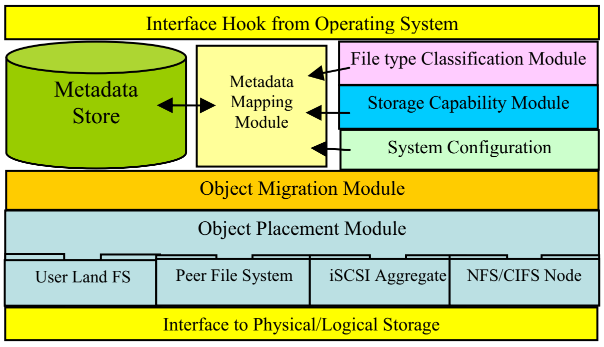

Figure 1. AOFS components

abstract layer hides the actual hardware from the operating system kernel. All AOFS modules are designed in pluggable manner. Different storage interface module can be added to provide access to different type of storage devices. AOFS provides interface to user land, iSCSI aggregator, NFS/CIFS node, and peer to peer file system implementation.

The use of each component in Figure 1 can be described in read and write operations. In a read operation, AOFS maps the file parameters to the target objects in the metadata store. The metadata mapping module is used to search the store. When metadata is found and interpreted, the location for each file object and file type is presented. With this information, AOFS can determine the best caching strategy before reading the data from the sub-system storage. In a write operation, files are first classified in the file type classification module. The disk allocation and the redundancy strategies are then selected from the indicated file type. The object placement module uses information from storage capabilities module to decide the best strategy for storing each file object. AOFS will segment a file into a number of data objects, which will be stored, based on AOFS tuning parameters. Then, the file object metadata is kept in the metadata store via metadata mapping module.

The Storage Capability module must understand storage capabilities of all partitions represented in an AOFS system. Examples include read/write access time, sequential/random write statistics, SMART information from a controller, and disk rebuilding status. Other modules can perform self-tuning by utilizing such real time information.

The Object Migration module works with the Storage Capability module and metadata information to dynamically tune the storage by migrating file object to the best possible storage tier.

In this paper, we emphasize the design of the file type classification module, which is important for dynamic tuning. From the previous research, the decision tree technique is often used to classify files based on their properties and attributes. A study in [6] shows that the classifying accuracy is in the 90% range. However, this technique is not adaptive to changes made to files. When a file type cannot be determined from a file name, a file print (file type footprint) is usually traced using the 1-gram Analysis. The technique utilizes the statistical analysis of binary content instead of a file parsing. A study in [7] shows the good results of using the technique to identify a set of unnamed files. In our work we are also interested in adaptive techniques for caching strategy selection. The study presented in [8] is thus explored. From the study, the hidden behavior of the application that generates an input/output access request stream is described using the Hidden Markov Model.

Our design extends the previous work by aiming at developing an efficient automatic file classification for AOFS based on an adaptive learning concept. The objective is to improve performance and accuracy based on experience (training). The module is divided into two parts which are static and dynamic classification. Static module is used to store new files into the system. The dynamic module is employed during run time to select caching strategy. Moreover, the module can be activated and deactivated by an administrator. The following sections describe our classification algorithms together with a list of attributes used to classify files.

II. File Attributes

In AOFS, the file attributes are kept in the metadata store. Table 1 shows the attribute list. These attributes are mainly used in our file type classification module.

AOFS also tracks a file access pattern, and updates the access information in the metadata store.

TABLE I
FILE ATTRIBUTES

Attribute Name	Description
fileObjList (ArrayList)	An array of file's objects - a file is segmented into objects according to the disk allocation strategies.
fileType (FileType)	Type of files, such as, text, video, and database files.
fileGroup (FileGroup)	Group of files that the classification module produces
fileNameSpace (NameSpace)	Location of a file returned from OS mapping
version (Revision)	Version or revision of a file
byteSize (long)	Total size of a file
owners (ArrayList)	A list of file owners: system or user
creator (Person)	A person who create a file
readers (ArrayList)	A list of users with a read permission
atime (Date)	Last access time
mtime (Date)	Last modified time
ctime (Date)	Last changed time
birthtime (Date)	File creation time
accessFreq1stTier (int)	First access frequency tier
accessFreq2ndTier (int)	Second access frequency tier
accessFreq3rdTier (int)	Third access frequency tier

III. Static Classification

When a new file enters the storage system, AOFS attempts to identify the file type using a series of techniques. The files attributes are then retrieved and the decision tree is employed to assign a file into a group. The results are all maintained in the metadata store. These results are used to select the primary disk allocation and caching strategies for the file. Figure 2 shows a flow chart of the overall steps.

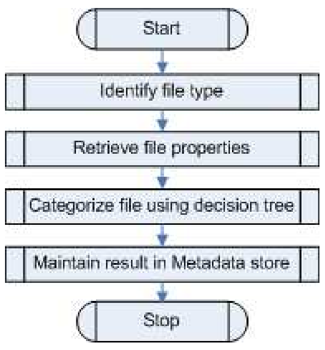
Start
Identify file type
Retrieve file properties
Categorize file using decision tree
Maintain result in Metadata store
Stop

Figure 2. Static classification flowchart

It is relatively straightforward for AOFS to retrieve the file properties, such as size, owners, readers, and creators. However, a file access sequence is not always obvious. Fortunately, the access sequence sometimes depends on a file type. For example, sequential access is usually observed in a video file, and random access is common in a database file. In order to gain such information AOFS provides a function to identify a file type as described in Figure 3.

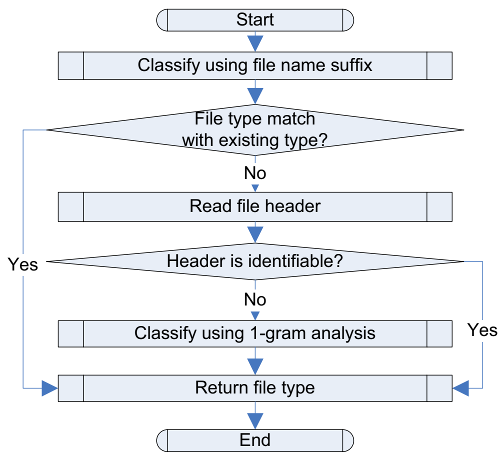
Start
Classify using file name suffix
File type match with existing type?
No
Yes
Read file header
Header is identifiable?
No
Yes
Classify using 1-gram analysis
Return file type
End

Figure 3. File identification flowchart

From Figure 3, AOFS uses a file name suffix (.txt, .jpg, .html, etc.) to identify a file type as an initial step. If a file name suffix is not complete or unknown to AOFS, a file header is read in an attempt to retrieve some file's specific information, such as a magic number or an ID string. Unfortunately, not all file types have such information. Furthermore, a file header can be intentionally modified by users. Thus, in a case where the header information is insufficient to identify the file type, AOFS employs the 1-gram analysis of byte value distribution.

Research in Fileprints suggests that each file type has a different byte distribution. Thus, using information in the first 20 bytes are enough to identify most file types. AOFS uses the 1-gram binary distribution of a file to compare with a set of example distribution types. The closet type is then selected. The advantage of the 1-gram analysis is not only identifying unknown file types, but also protecting AOFS against infected files. Viruses may be detected if they do not conform to any of the file type distribution.

Once the file type is identified, the decision tree can be used to categorize the file according to its properties and then assign it into a prioritized group. AOFS employs the decision tree technique as it is simple and efficient. The tree used is shown in Figure 4.

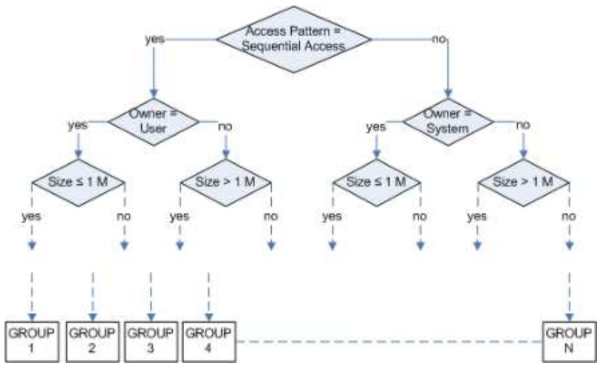

Figure 4. AOFS decision tree

From Figure 4, there are N groups of files. For instance, GROUP1 has the following attributes: owner is a regular user, sequential access is observed, and the file size is less than 1 MB.

These file groups are then assigned a different priority levels, which will be used to select adaptive policies. For example, files in a high priority group will be stored in a high performance storage device in order to guarantee a high speed access. Files in a low priority groups may be stored remotely. When a disk is full, lower priority files may be segmented into several objects and migrated to a lower performance or a remote storage.

IV. Dynamic classification

Dynamic classification aims to enhance the efficiency of caching and dynamic tuning during runtime. The file access time can be reduced by object pre-fetching based on a previously analyzed access pattern. An adaptive learning technique, called Hidden Markov Model (HMM) is employed in AOFS.

During runtime, AOFS collects file access information, such as, access sequence and frequency, last access time, last modified time, and last changed time in the metadata store.

The process is similar to the static classification. However, the file identification function is not used. In stead, AOFS provides a finer classification by considering previously observed access pattern. For example, information on creation, last access, and last modified time together with access frequency are used to estimate the status of a file in its life cycle.

Adaptive caching strategies are selected based on input/output access patterns, which can be unavailable from observing a short sequence of input/output requests. Thus, during an execution, AOFS keeps a log on file access for a long period of time. This log file can then be used to train the HMM. The trained model can later be employed to select the most possible access sequence and file objects can be pre-fetched accordingly.

The training process of the HMM maintains counts of objects' read or write transitions. The probabilities for each transition are calculated by dividing the number of occurrences of the transition by the total number of transitions from each object. Then, the optimal sequence can be selected from the most probable path.

For example, Figure 5 illustrates an HMM that models a sequential read pattern for a file with five objects.

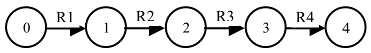

Figure 5. A model describing sequential reads

Example in Figure 6, on the other hand, shows a conditional branch in the underlying program. From the reading in object 0, the program either reads the rest of the file sequentially or skips object 3. The HMM computes the probabilities for both transitions. AOFS then decides whether object 3 should be pre-fetched.

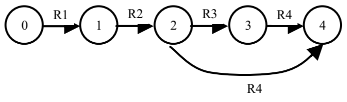

Figure 6. A model describing two possible paths

The File Type Classification module can automatically learn how to classify files using information in the metadata store. The classification results are used to automatically select adaptive storage policies in AOFS. Furthermore, the storage policies can also be manually specified by an administrator of the system.

V. Conclusion

This research presents an alternative file system framework based on object-oriented concept. The File Type Classification module of the framework is described in detail. The file attributes and access pattern are used in both static and dynamic classifications to provide both efficiency storage and caching. This paper provides a status report on our progress in developing the concepts. The performance evaluations of the new framework will be conducted in the near future.

Acknowledgment

The authors would like to thank Throughwave (Thailand) Co.,Ltd staffs for their helpful discussions and on Adaptive Object File System framework.

References

[1] H.L. John and P.A. David, *Computer Architecture A Quantitative Approach, 3rd ed.*, Morgan Kaufmann Publisher, United State of America, Pages: 2-4, 678-787.

[2] W.W. Hsu and A.J. Smith 2004, “The performance impact of I/O optimizations and disk improvements”, IBM J. Res. Develop.

[3] W. B. Ligon III and R. B. Ross, “Implementation and Performance of a Parallel File System for High Performance Distributed Applications”, Parallel Architecture Research Lab, Clemson University, 1996.

[4] G. James and A. Randy, “The Design, Implementation, and Evaluation of a Predictive Caching File System”, Department of Computer Science, University of Kentucky, USA, 1996.

[5] O. Gorter, “Database File System – An Alternative to Hierarchy Based File Systems”, University of Twente, the Netherlands, August 2004.

[6] M. Michael, T. Eno, G.R. Gregory, E. Daniel, and S. Margo, “File classification in self*- storage systems”, in proceedings of the First International Conference on Autonomic Computing (ICAC-04), May 2004.

[7] L. Wei-Jen, W. Ke, S.J. Salbatore, and H. Benjamin, “Fileprints: Identifying File Type by n-gram Analysis”, Proceedings of the 2005 IEEE Workshop on Information Assurance, United State Military Academy, West Point, NY, June 2005.

[8] M.M. Tara and R.A. Daniel, “Input/Output Access Pattern Classification Using Hidden Markov Models”, ACM Computing Surveys, 1997.