Coordinating Parallel Hierarchical Storage Management in Object-base Cluster File Systems [*]

Dingshan He, Xianbo Zhang and David H.C. Du
Department of Computer Science and Engineering
DTC Intelligent Storage Consortium (DISC)
University of Minnesota
{he,xzhang,du}@cs.umn.edu

Gary Grider
Los Alamos National Laboratory
Department of Energy
ggrider@lanl.gov

Abstract

Object-based storage technology enables building large-scale and highly-scalable cluster file systems using commodity hardware and software components. On the other hand, a hierarchy of storage subsystems with different costs and performance should be incorporated into such systems to make them affordable and cost-effective. Existing SAN-based (block-based) cluster solutions suffer from slow data movement between storage levels due to their single-archiving-point architecture. In this paper, we propose a novel parallel data moving architecture in object-based cluster file systems. Data movements are coordinated and performed in parallel between multiple pairs of storage subsystems. In addition, data movements are fully automated and transparent to users. Our proposed parallel data moving architecture is prototyped on the Lustre file system. Performance study shows that our scheme can scale up easily by adding more pairs of hierarchical storage devices.

1. Introduction

High performance computing (HPC) community has realized that a Scalable, Global and Secure (SGS) file system and I/O solution is necessary since late 90s [1]. Object-based Storage Device (OSD) is a potentially promising approach for building such a file system. Receently two solutions based on the concept of OSD have emerged: the Lustre file system [2] and the Panasas ActiveScale file system [3]. These OSD-based systems are expected to serve clusters with 10,000's of nodes, 100's GB/sec I/O throughput and several petabytes of high performance storage [2]. Due to the large amount of data generated within the HPC environment [1], data archiving and Hierarchical Storage Management (HSM) are necessary to reduce the total storage cost and to assure the sustained high performance in the system. High performance data movement between subsystems of storage hierarchy is extremely important for this type of cluster file systems. A storage hierarchy is usually composed of expensive SCSI storage arrays for high performance accessing, cheaper SATA storage subsystems for data staging and high-capacity robotic tape libraries for data archiving. The movements of data between storage hierarchy are made transparent to applications through HSM software. HSM takes advantage of the fact that data are not of equal importance during any given period of time for applications. Typically, only a small subset of the entire data set is actively used by applications.

Archive systems have been identified as one possible bottleneck in a SGS file system and scalable HSM solutions are declared as extremely desirable [1]. The expected archival storage bandwidth was 10GB/sec (over 35 TB/hr) in 2003 and 100GB/sec in 2006 [4]. However, the backup and restore records announced by SGI and its partners in July 2003 was only 2.8GB/sec (10.1TB/hr) for file-level backup and 1.3 GB/sec(4.5 TB/hr) for file-level restore, even through these already tripled the previous record in 2003. Obviously, the existing solution is far from catching up with the growing requirements of archival storage bandwidth. Filling the gap between the desired archiving speed and the currently available speed is our research focus.

A high I/O speed for moving data between storage hierarchy would be appreciated by scientific computing and business users. Scientific simulations at DOE usually run for days or months so the system must be prepared for software and hardware failures during the running period. The practice is to dump the simulation states at every checkpoint to persistent storage so that simulation can resume from the latest checkpoint after failure. It is possible that multiple terabytes of data are generated over a short period

[*]This project is supported by a grant from Los Alamos National Laboratory.

of time (every 30 minutes as described in [1]). The total amount of data can easily exceed the storage capacity provided by the disk-based storage and have to be moved to low-cost storage media like tapes. The data moving speed therefore has a big impact on the simulation performance since simulation process has to wait for the completion of the data moving before it can resume. To reduce the financial costs for maintaining large clusters, high performance computing power and storage resource are usually shared by various applications among different researchers. Given a specific time, only a subset of the applications and their associated data need to be retained on the expensive, high performance storage. The in-active data can be moved to low-cost storage media. The data moving speed between subsystems of storage hierarchy directly affects the overall system performance. Due to the economy globalization, business data need to be accessed 24 hours a day, seven days a week. For business continuance, data are periodically backed up to low-cost tape media. Should any data disaster happen, the system restore time needs to be minimized to reduce the business financial loss.

In this paper, we propose a novel parallel data moving architecture for the high performance object-based cluster file systems. The aforementioned archive/restore bottleneck in the current systems is overcome by enabling multiple parallel data movements between multiple pairs of storage subsystems of different storage hierarchies. The object-based storage interface is an enabling technology in that it allows direct data movement between storage subsystems and avoids the performance bottleneck of any single subsystem. Further more, each OSD has a local instance of HSM software that is responsible for the data movements between that OSD and one or more designated lower-level storage subsystems. The major challenge is to coordinate the multiple instances of HSM on different OSDs to

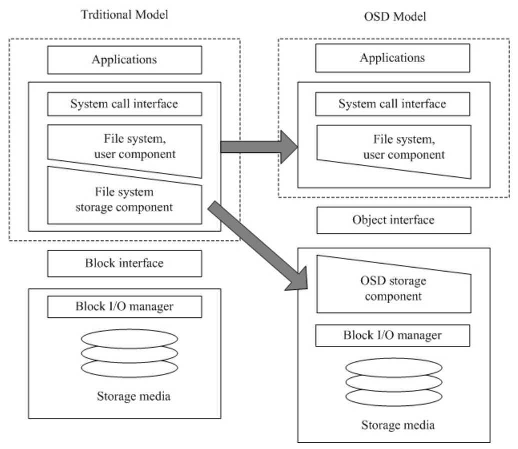


Figure 1. Comparison between traditional file systems and OSD [5]

achieve real parallelism in data movements since multiple data objects associated with the same application can be located in different OSDs. Another challenge comes from possible component failures that are faced by any cluster storage systems. We have addressed both challenges in our proposed parallel data moving system.

The rest of this paper is organized as follows. Section 2 introduces some background information. Our proposed parallel data moving architecture is presented in Section 3. Then, we discuss how to maintain consistency among distributed components and how to handle component failures in Section 4. In Section 5, we discuss the prototyping of our proposed scheme on the Lustre file system. The performance data of our prototyping is also reported in the same section. Several related works are presented in Section 6. Finally, we conclude our contributions in Section 7.

2. Background

In this section, we discuss three concepts that are essential to our work: object-based storage interface, object-based cluster file systems and hierarchical storage management.

2.1. Object-based Storage Interface

Object-based storage is an emerging storage interface standard designed to replace the current storage interfaces, mainly SCSI and ATA/IDE. It is motivated by the desire to design a storage architecture providing 1) strong security, 2) data sharing across platforms, 3) high performance and 4) scalability [6]. The storage architectures in common use today based on SCSI and ATA/IDE interfaces are direct-attached storage (DAS), storage area network (SANs), network-attached storage (NAS) and a newer architecture called SAN file systems. Unfortunately, none of these four architectures achieves the aforementioned four desirable features of an ideal storage architecture at the same time. Therefore, system designers usually first decide which of these features are more important than others as a trade-off in choosing a storage architecture.

Objects are storage containers with a file-like interface in terms of space allocation and data access. An object is of variable-length and can be used to store any types of data, such as traditional files, database records, images or multimedia data. Objects are composed of data, user-accessible attributes and device-managed metadata.

A device that stores objects is referred to as an *Object-based Storage Device* (OSD). OSDs can be in many forms ranging from a single disk drive to a storage brick that contains a storage controller and an array of disk drives. The storage media can be magnetic disks, tapes or even read-only optical media. Therefore, the essential difference between an OSD and a block-based device is their interfaces instead of the physical media. The most immediate effect of an object-based storage device is the offloading of space management from storage applications. The comparison between the traditional model on block I/O and the OSD model is presented in Figure 1. The storage component (managing storage space including free space management and mapping logical storage objects such as files to physical storage blocks) of block-based file systems is offloaded into OSDs in the OSD model. This improves data sharing across platforms as the platform-dependent metadata management is now inside OSDs. OSD model also improves scalability since hosts running storage applications no longer need to coordinate metadata updates and the data path and control path are separated. Furthermore, the OSD model can provide strong security at per-object basis by letting the device enforce security checks on each access request.

2.2. Object-based Cluster File Systems

Using OSDs as building blocks, Figure 2 shows a way to construct object-based cluster file systems (OCFS) [2, 3]. A metadata server (MDS) is used to manage metadata that include hierarchical or global namespace, file attributes and file-to-object mapping information. Note that such metadata are only part of the entire metadata used by file systems of block-based devices. Most noticeably, the metadata server in OCFS does not need to manage file-to-block mapping information and does not need to keep track of free space on storage devices. As presented in the previous section, both functions are offloaded to OSDs. The metadata server is a logically centralized component (physically it can be a cluster of servers) to expose a shared and uniform namespace to clients of OCFS. With many of metadata management functions taken care by either the metadata server or OSDs, clients are only responsible for in-memory data structures such as inodes and dentries of traditional file systems of block-based devices. Since client machines of OCFS are typically also application servers such as Web servers or DBMS servers, the light-weighted file system functions allow them to serve their applications more efficiently.

The MDS in Figure 2 also shows the optional function of security manager. In the previous section, we mentioned that OSD model can provide strong security at per-object basis by letting OSDs perform security checks on every access request. In order to do this, every access request should be accompanied with a tamper-proof message called credential describing what operations this request is allowed and disallowed. The security manager is the authority in this architecture to maintain such information

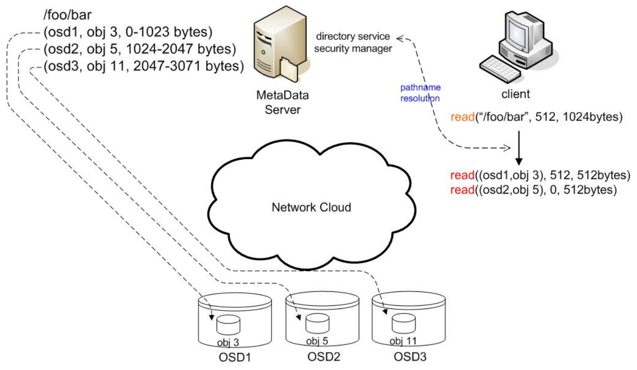

/foo/bar
(osd1, obj 3, 0-1023 bytes)
(osd2, obj 5, 1024-2047 bytes)
(osd3, obj 11, 2047-3071 bytes)
MetaData Server
directory service
security manager
pathname resolution
client
read("/foo/bar", 512, 1024bytes)
read((osd1,obj 3), 512, 512bytes)
read((osd2,obj 5), 0, 512bytes)
Network Cloud
obj 3
obj 5
obj 11
OSD1
OSD2
OSD3

Figure 2. Object-based cluster file system

about who can do what on a particular object. It generates the credentials upon the request of authenticated clients.

Figure 2 also illustrates how an OCFS employs file striping to handle IO-intensive and data-intensive accesses to large files. Any logical file can be striped into multiple objects on multiple OSDs. This file-to-object mapping information, referred to as the layout information, is part of the metadata managed by the MDS for every file. When a client opens a file, its layout information is extracted from the MDS and stored in the file's in-memory inode structure on the client along with other metadata information of the file. Then, the client can use simple calculations based on the layout information to convert file access requests to one or more object access requests. When there are multiple object access requests, they are performed in parallel to multiple OSDs instead of in sequence. The benefits of parallel accessing are even more obvious when there are multiple concurrent clients with multiple concurrent processes. With a typical random access pattern, traffics can be evenly spread across OSDs. Similar benefits apply to a set of small files that are not striped when they are spread among OSDs.

2.3. Hierarchical Storage Management

Hierarchical Storage Management (HSM) is a policy-based management of file backup and archiving on a hierarchy of storage devices. The users need not to know when files are being archived and the access request may retrieve data from online or lower-level storage devices. Using an HSM software, an administrator can establish guidelines for conditions under which different kinds of files are to be copied or migrated to a lower-level storage device. The conditions are typically based on ages of files, types of files, popularity of files and the availability of free space on storage devices. Once the guideline has been set up, the HSM software manages file migrations automatically.

HSM makes the file migrations transparent to file system users by using two mechanisms: *stub files* and *data* *management API* (DMAPI). A stub file is a pseudo file kept in the file system after the file data are migrated to lower-level storage. Because of it, file system users will not see a file missing although it has been migrated to a lower-level storage device and no longer available on online storage devices. More importantly, the extended attributes of a stub file are used to point to the real location of the file in the lower-level storage. DMAPI is the interface defined by Data Storage Management (XDSM) specification [7] that uses events to notify user space Data Management (DM) applications about operations on files. HSM software is one kind of DM applications. When file system users read/write a stub file, the kernel component of XDSM generates read/write event and notifies HSM who should have registered through DMAPI as willing to receive such kind of events. The process of reading/writing the stub file is blocked until a response is received from HSM. The HSM software handles read/write events on stub files by migrating the requested file back to online storage so that the stub files turn into real files. A response is sent through DMAPI after the migrations completed. The blocked read/write process can then proceed to perform the requested operation on the real file.

3. Parallel Archiving Architecture

There is no doubt about the necessity of a storage hierarchy in large-scale storage systems. As the object-based cluster file systems like ActiveScale and Lustre have demonstrated their extraordinary performance and scalability, it is natural to ask how to incorporate some kind of HSM into OCFS while avoiding HSM becoming a new bottleneck of the system. Various HSM solutions exist for single-server platforms and block-based SAN file systems. In Section 3.4, we compare our proposed architecture with such solutions to show that they cannot take fully architectural advantages of the potentials of OCFS. In this section, we start with analyzing the architectural advantages of OCFS and motivate serval key design choices. We then present our proposed parallel data moving architecture in OCFS. Several key operations are also elaborated to explain how the system works. Finally, we briefly compared our proposed architecture with existing solutions in single-server platforms and block-based SAN file systems.

3.1. Design Rationale

We have identified the following three architectural features that contribute to the high-performance and high-scalability of OCFS the most:

- Feature 1: function offloading
  The free-space management functions are offloaded to OSDs such that clients and MDS become light-weighted and can perform efficiently. In addition, the communication cost for free-space management is also reduced from the communication network. This additional management task causes very little problem since OSDs typically have more than enough processing power and memory buffer beyond the requirement of handling data transfers and physical location mapping in block-based storage devices.

- Feature 2: separated control and data paths
  As MDS is dedicated for metadata management, it performs and scales better than having to handle both metadata and data. On the other hand, MDS is no longer in the critical path for clients accessing data from OSDs. Therefore, initiated data accesses are not affected by ongoing metadata requests.

- Feature 3: parallel data paths
  With proper configuration of the communication network, e.g., using FC switch instead of FC-AL, multiple parallel data paths from clients to OSDs are realized. The bandwidth of the overall system can be scaled up easily by properly adding more switches into the network.

In our proposed parallel data moving architecture, we fully explore the aforementioned features of OCFS and carefully avoid undermining achieved performance and scalability before introducing the required hierarchical storage management functions. Following is a list of the key architectural features of our proposed parallel data moving architecture for OCFS:

- embedded HSM component on OSD
  We propose to add one more task, a HSM component, to OSDs. This should not be a problem in terms of the capability of OSDs considering that Lustre and ActiveScale are using general-purpose CPU with a decent amount of memory in their OSD targets. The local HSM instance on an OSD performs similar to today's HSM products in single-server platforms. It has a daemon periodically checking the availability of free-space in the OSD's online storage and initiating data migrations when conditions meet. On the other hand, when an OSD receives data access of an object not in its online storage, the local HSM is invoked to retrieve the object from a lower-level storage. This design choice is an extension of the Feature 1 of OCFS. It avoids building a complicated and centralized HSM component, which is doomed in terms of performance and scalability.

- direct data migration paths
  For one migration operation, the object data is transmitted on the SAN at most once - directly from the

OSD to its associated lower-level storage. It is possible that the migrated data is not transmitted over the SAN at all if the configuration chooses a directly attached secondary storage for the particular OSD. In comparison, the migrated data has to be transmitted twice in block-based SAN file systems - once from the online storage device to the HSM host's memory and once from the memory to the lower-level storage device.

- parallel data migration paths
  Like Feature 2 of OCFS, data migration paths are parallelized with proper configuration of SAN. To explore this feature, it is better to use many smaller storage systems as lower-level storage than use a few large storage systems. In the former case, each OSD can have a dedicated lower-level storage system or a few number OSDs share one. In contrast, in the latter case many OSDs share one lower-level storage system so that the data migration paths may collide at the limited bandwidth of the shared lower-level storage systems. Economically, it is also cheaper to buy a bunch of smaller storage systems than buy a large storage system of equal capacity. From scalability point of view, it is also easier to expand by adding new fabric switches and small storage systems gradually than buying an expensive large system that may not fully utilized at the moment.

- separated migration coordination paths
  Very often, the distributed local HSM instances on different OSDs require coordination among themselves. Although they are physically independent from each other, the data objects that they are maintaining may not be always independent. Especially, when file striping is used to handle IO-intensive and data-intensive requests, an object is only a strip of the file. Such an object is called a *striped object* in this paper. Its fellow striped objects are hosted by other OSDs under the management of their local HSM. Several non-striped data objects on different OSDs may also be required for a given application at the same time. In order to explore the feature of parallel data movement paths, one policy can be letting striped objects or associated objects always migrate in parallel. We choose to separate the coordination control path from the migration data path following the spirit of feature 2 of OCFS.

- versatile storage interfaces
  In the context of OCFS, the most natural choice of storage interface for lower-level storage subsystems is object-based storage interface. However, we decide not to exclude the possibility of using the other two major storage interfaces, i.e., block interface and NAS. The consideration here is that organizations may already have major investments using non object-based storage interface. The trick is to let the OSD's local HSM use a universal file system interface to access its lower-level storage subsystems. If the lower-level storage subsystems use a block-based interface or a NAS, the local HSM can still communicate with them. Of course, the performance will be different depending on the choice.

- eliminating DMAPI
  As discussed in Section 2.3, DMAPI/XDSM is widely used by many HSM products. Unfortunately, DMAPI/XDSM is too heavy and HSM softwares only use a small number of events specified by DMAPI. Experiences have also shown that DMAPI/XDSM does not scale well. This is also why popular file systems and operating systems are reluctant to support DMAPI/XDSM. For better performance and scalability, we have managed to use file attributes in MDS to keep track of where file objects are currently stored and use local HSM to trap the events. The result is that DMAPI is no longer needed in our architecture. In addition, we also do not need to leave stub files in the OSDs' local file systems since they are not part of the namespace visible to OCFS users.

3.2. Architecture Overview

Figure 3 illustrates our proposed parallel data moving architecture in OCFS. In order to coordinate distributed local HSM instances, we introduce three new components into OCFS: *archival attributes*, a *Migration Coordinator* and *Migration Agents*. Each of them is elaborated in the follows.

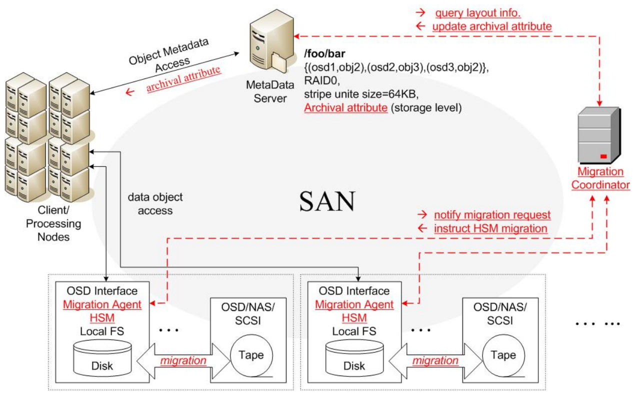


Figure 3. Parallel archiving architecture in OCFS

3.2.1. Archival Attributes

In MDS, each data file (not directory files) has a new archival attribute. It is a value simply indicating the level of storage hierarchy that the file data is currently stored. The possible number of levels of storage hierarchy is a system parameter, which should be supported by all participating local HSM instances. For example, it can be as simple as only level 0 indicating disk storage and level 1 indicating tape storage, or having other levels like SCSI disk level and SATA disk level. Currently we are assuming a fixed storage hierarchy. Of course, if a disk storage may be associated with more than one tape devices, the archival attribute needs to be more complicated than the current design. For non-striped files, it is obvious what this archival attribute means. For files with striped objects on multiple OSDs, this single archival value is still valid since our scheme is designed to collaborate the archive and recall behaviors of involved local HSM instances. The object stripes should be at the same level of storage hierarchy indicated by the archival attribute of the file. The values of archival attributes are updated by requests from the Migration Coordinator when there are object migrations. The functions of the Migration Coordinator are going to be discussed later in this section. When accessing data from OSDs, a client sends the request together with the archival attribute of the file. How OSDs use archival attributes is going to be discussed momentarily.

3.2.2. Migration Agent

There is a Migration Agent component in each OSD and it performs the following three major functions: 1) taking care of the functions typically provided by DMAPI to enable non-DMAPI local file systems in OSDs; 2) interacting with the local HSM instance; 3) interacting with the remote Migration Coordinator. OSDs receive client object requests that are accompanied with their corresponding archival attributes. The OSD interface converts the client request to a request of its local file system. Then the file system request and the archival attribute are passed to the Migration Agent. The archival attribute specifies the level of storage hierarchy that the data is currently stored. If it indicates the online storage, the Migration Agent just passes the file system request to the local file system that servers the request directly. However, if the archival attribute specifies a lower-level storage, the Migration Agent interacts with the remote Migration Coordinator to ask for collaboration of other HSM instances on different OSDs if the requested data object is just a stripe of the requested file. Under the coordination of the Migration Coordinator, all involved OSDs perform data movements in parallel. In another situation, the local HSM instance may decide to migrate some data in the primary storage to the secondary storage to make room in the primary storage (when the available free space decreases below a preset threshold). The local HSM instance interacts with and notifies the Migration Agent with the list of migrating objects. The Migration Agent will again contact the Migration Coordinator for possible collaborations.

3.2.3. Migration Coordinator

The Migration Coordinator receives archive/recall notifications from Migration Agents. For data objects corresponding to non-striped files, the Migration Coordinator's responsibility is only to notify the MDS to update the archival attributes of the corresponding files. However, for data objects belonging to striped files, the additional responsibility of the Migration Coordinator is to coordinate the achive/recall by instructing multiple local HSM instances on different OSDs to perform data migrations of all striped objects. In order to do this, the Migration Coordinator needs to contact the MDS to get the layout information of striped files. The Migration Coordinator instructs local HSM instances by contacting the Migration Agents who in turn contact their local HSM instances, instead of directly contacting them.

3.2.4. Interfaces Between Modules

Modules in Figure 3 interact with each other through well defined interfaces. There are three pairs of new interactions introduced in our proposed architecture: MDS-MC (Migration Coordinator), MA (Migration Agent)-HSM and MC-MA. If the two entities are on different hosts like the MC-MA pair, the interface is implemented through RPC (Remote Procedure Call). Otherwise, the interface is implemented through exported kernel module APIs.

The MDS-MC interface between the metadata server and the migration coordinator is defined as following:

- Metadata Server exported APIs
  - *GetLayout*: This API is used by MC to query the layout information of the file containing a certain data object.
  - *SetArchAttr*: This API is used by MC to update a file's archival attribute.
- Migration Coordinator exported APIs
  None since MDS does not need any help from MC.

The MA-HSM interface between a Migration Agent and a local HSM instance on the same OSD is the bridge between the HSM instances and the rest of the parallel data moving architecture. It is important for that any non-parallel commercial HSM product can be incorporated into our propose parallel archiving architecture as long as it is compliant with this interface. The MA-HSM interface is define as following:

- Migration Agent exported APIs

- *ReqArchive*: HSM uses this API to report its intention of migrating a list of objects from the online storage to the lower-level storage in order to free some online storage space.

- HSM exported APIs
  - *HSMArchive*: This API is used by MA to instruct HSM to migrate an object from the online storage to the lower-level storage.
  - *HSMRestore*: This API is used by MA to instruct HSM to migrate an object from the lower-level storage to the online storage.

Finally, the MC-MA interface between the Migration Coordinator and the Migration Agents is define as following:

- Migration Coordinator exported APIs
  - *ReqAchive*: This API is used by MA to report HSM's intention of migrating a list of objects from the online storage to the lower-level storage.
  - *NotifyRestore*: This API is used by MA to report its intention of restoring an object that is not in online storage and has an access request from some client.
- Migration Agent exported APIs
  - *ArchiveObject*: This API is used by MC to instruct MA to archive a list of objects.
  - *RestoreObject*: This API is used by MC to instruct MA to restore an object.

3.3. Key Operations

In order to help understand how the components work together in a running system, we elaborate two key operations: 1) a client accesses objects not in OSD's online storage and 2) a local HSM instance archive objects to free its OSD's online storage space. Note that both descriptions omit steps related to concurrency control and error recovery for easier understanding. Section 4 explains the concurrency control and error recovery mechanisms to guarantee consistency under distributed concurrent clients and local HSM instances and under component failures.

Figure 4 is the sequence steps of a client accessing an object not in OSD's online storage. Following is the explanation of the steps:

1. The client retrieves the archival attribute of the file being accessed.

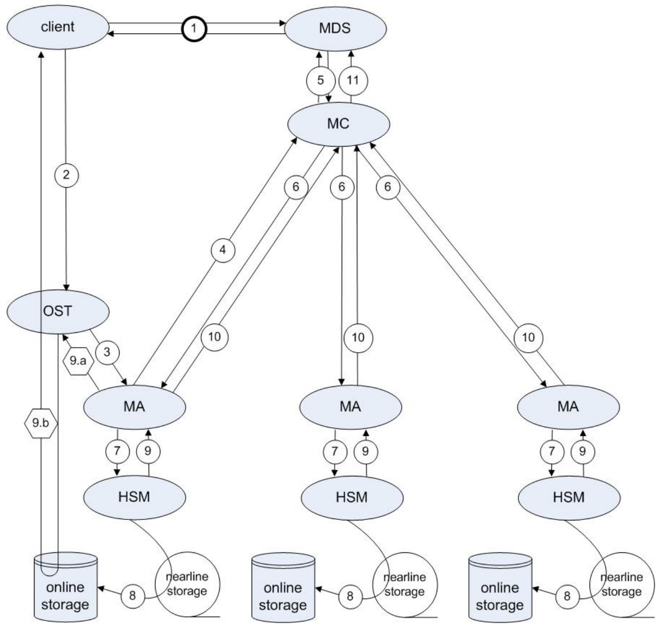


Figure 4. Accessing object not in online storage

2. The client translates the file access request to one object access request. Then, it sends object access request along with the file's archival attribute to the OST module of the OSD hosting the requested object. The translation can result in accessing multiple objects but we do not show such case here.
3. The OST module translates the object access request to the back-end file system access request and passes the result along with the archival attribute to MA.
4. The MA finds out the requested object is not in the online storage by checking the archival attribute. It then execute *NotifyRestore* RPC of MC.
5. The MC calls *GetLayout* to retrieve layout information of the requested file from the MDS.
6. The MC calls *RestoreObject* RPC to every MA of the OSDs hosting the striped objects of the requested file in parallel.
7. The MAs calls *HSMRestore* functions of their local HSM instance to restore their striped objects from the lower-level storage to the online storage in parallel.
8. Each local HSM instance restores the object in parallel.
9. Each local HSM instance responds to its MA to indicate finishing of restoring. For the OSD initiating the

restore process, which is the left-most OSD in Figure 4, the requested object is ready in the online storage at this moment. Therefore, the following steps are performed asynchronous with the rest of the main sequence:

  (a) The MA notifies the OST to retrieve data from the online storage.
  (b) The OST performs normal access operation on the requested object in the online storage.

10. Each involved MA sends response to the MA to indicate the finish of restoring.

11. The MC calls *SetArchAttr* to update the archival attribute of the file in the MDS.

Figure 5 is the sequence steps of a local HSM instance migrating objects from the online storage to the lower-level storage. The steps are elaborated as follows:

1. The HSM daemon in the left-most OSD in Figure 5 finds out that the available free-space in the online storage has decreased below a predefined threshold. It walks through the objects in the online storage to prepared a list of objects to be migrated to the lower-level storage. Then, the HSM instance calls *ReqArchive* to report its intention to its local MA.

2. The left-most MA in Figure 5 calls *ReqArchive* RPC to the MC to relay the report of migration intent of its local HSM instance.

3. The MC calls *GetLayout* to retrieve the layout information of the files containing the requested objects from the MDS.

4. The MC calls *ArchiveObject* RPC to every MA of the OSDs hosting the striped objects of the requested files in parallel.

5. The MAs calls *HSMArchive* functions of their local HSM instance to migrate a list of objects from the online storages to their lower-level storage in parallel.

6. Each local HSM instance migrates the requested objects in parallel.

7. Each local HSM instance responds to its MA to indicate finishing of data migration.

8. Each involved MA sends response to the MA to indicate the finishing of data migration.

9. The MC calls *SetArchAttr* to update the archival attributes of the files in the MDS.

10. The MC responds to the initiating MA to indicate the finishing of migration.

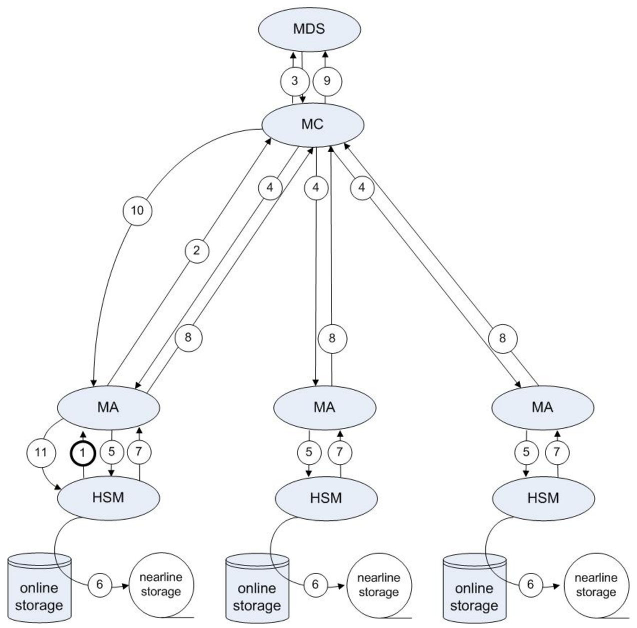


Figure 5. Migrating object from online storage to lower-level storage

11. The initiating MA responds to its local HSM instance to indicate the finishing of migrations. The local HSM instance can check the available free-space level again and initiate more migrations if necessary.

3.4. Comparison to Single-server and Block-based SAN solutions

The obvious drawback of HSM solutions on single-server platforms is that the lower-level storage can not be shared by other servers. Such solution cannot scale up well as the demands increase. In enterprises, it is also difficult to justify purchasing expensive tape libraries without sharing them among multiple hosts.

HSM solutions in block-based SAN environment such as SGI DMF [8] and IBM HPSS [9] enable the sharing of expensive tape libraries. However, limited by the capability of block devices used in these system, migration operations require data to be transmitted on the SAN twice: once from online block devices to memory of migration hosts and again from migration hosts memory to lower-level storage devices. Due to the limited number of migration hosts typically setup in such systems, they are overwhelmed with the responsibility of migrating data for lots of storage systems. In comparison, our proposed architecture explores the capability of OSDs. The effects are that data only need to be transmitted over SAN once and migration tasks are more widely distributed among OSDs.

	ALock	**BLock**	**RLock**
ALock	CP	EX	EX
BLock	cancel	CB	error
RLock	PR	EX	CB

Table 1. Lock compatibility table

Another limitation of the existing HSM solutions in block-based SAN environment is that they emphasize sharing of archiving storage too much so that they tend to use only a few big archiving storage subsystems shared throughout the entire system. The problem arises from the limited bandwidth to access the archiving storage systems. This results in long archive and restore time. Considering the shrinking backup window today, this becomes more a concern for organizations using such systems. In comparison, our architecture is motivated by solving this narrow-pipe limitation of archive and restore. In stead of using a few large archive storage systems, we choose to use many smaller ones and properly connect them to online storages using switch-base fabric. Multiple archive/restore data paths exist thus archives/restores can be executed in parallel.

4. Consistency and Error Recovery

Like any other distributed systems with concurrent operations, we have to guarantee data consistency in spite of concurrent object access operations, archive operations and restore operations. Also, we have to guarantee data consistency in case of possible component failures. In this section, we first present a specially designed locking mechanism to handle concurrency control of object access, archive and restore operations. Then, we describe a logging-based mechanism for error recovery.

4.1. Migration locking mechanism

In our proposed architecture, the critical data structures that needs to be protected against concurrent accesses are the *archival attributes* of files in the MDS. Clients read archival attributes when they are going to access data objects of the files. The migration coordinator updates archival attributes when it has finished archive or restore migrations of the files' striped objects.

Although it looks like a classic read-write locking scenario, it is actual not. For example, if a client is accessing a file whose data is on lower-level storages, it anyway requests a read lock on the archival attribute. However, when the MC try to request a write lock on the same archival attribute in order to restore the file's objects to serve the request, it will be blocked by the early read lock and it is a deadlock. In addition, our specific migration semantics make read-write locking not appropriate. Imagining that a file's archival attribute has been read locked by a client for data accessing. It is possible that one OSD hosting a striped object of the file tries to archive the object in the meantime. In traditional locking semantics, the write lock request by the MC will be blocked until the release of the read lock by the client and then continue the archive. However, this is not desirable since we are backing up a file that has just been accessed. According to the locality property of data accessing, the right way should be keeping the objects of the file on online storages.

With the aforementioned properties in mind, we have designed a specialized locking mechanism for concurrency control of archival attributes. There are three types of locks: Access Locks (ALocks), Backup Locks (BLocks) and Restore Locks (RLocks). Table 1 illustrates the compatibilities between locks. Note that these locks are asymmetric in the sense that their compatibilities and corresponding actions depends on the sequence of locking requests. For a pair $(xlock, ylock)$ in Table 1, the column element $ylock$ is the existing lock type on an archival attribute and the row element $xlock$ is the newly requested lock type. CP indicates the lock types are compatible in that requesting sequence. $(ALock, ALock)$ has value CP since clients accessing files do not modify archival attributes by themselves. EX indicates the new $xlock$ is incompatible with the existing $ylock$ on the archival attribute and the requesting process of the $xlock$ should be blocked until the release of $ylock$. According to Table 1, clients requesting $ALock$ on archival attributes should be blocked if there are existing $BLock$ or $RLock$ on the requested archival attributes. This is because clients should not be allowed to access the file during the process of migrations. Also, a request for $RLock$ is blocked if a $Block$ exists since restore should not be allowed during the process of archive. However, the reverse case, i.e., a archive request occurs during the restore process as indicated by $(BLock, RLock)$, is an erroneous case since it is impossible for OSDs to initiate archive process for an object not on the online storages. PR for $(RLock, ALock)$ indicates that a request for $RLock$ can preempt existing $ALocks$ on the archival attributes. This avoids the deadlock situation described earlier in this section. The preemption does not cancel the $ALocks$ already granted to clients. Instead, before the releasing of $RLock$, a message is sent to every owner of $ALocks$ to tell them that the archival attributes that they got earlier have been updated in the MDS. CB indicates that $xlock$ is intending to do actually the same thing as the existing $ylock$. Therefore, it can just let the owner of $ylock$ to finish the task and notify the owner of $xlock$ through asynchronous callback. This happens when more than one independent OSDs request to archive or restore the same file objects as indicated by elements $(BLock, BLock)$ and $(RLock, RLock)$. The asynchronous callback method is used instead of blocking since archive requests issued by local HSM instances contain a list of objects. It is unnecessary to block archive operations of other files. Finally, we use an optimization based-on local property in the case of $(BLock, ALock)$. An archive request for a file being accessed is cancelled. This could happen when a client is accessing one of the striped objects of a file and another striped object's hosting OSDs wants to archive it to free online space.

This migration locking mechanism is implemented in the MDS. *ALocks* are requested by clients in the Step 1 of Figure 4 and released at the end of the file access. *RLocks* are requested by MC in the Step 5 of Figure 4 and released as part of MDS processing of Step 11. Finally, *BLocks* are requested by MC in the Step 3 of Figure 5 and released in the processing of the Step 9 of Figure 5.

4.2. Error recovery

Component failures can cause the system to enter an inconsistent state without proper error recovering mechanism. For example, in the sequence diagram of Figure 5, imagine the right-most OSD fails after Step 4 and before Step 8. Even if that OSD can switch to an fail-over processor and restart functioning, the system will not be in a consistent state unless the OSD can finish the archive operation and response to the migration coordinator as in Step 8.

The scheme that we have employed is part of the error recovery framework of OCFS. We explicitly assume that the OCFS has capability to discover component failures and system restarts, typically through heartbeating signals and rebooting sequence numbers. We also assume OSDs have fail-over processing components.

Not surprisingly, we use logical logging (also known as journalling in file systems) and replaying to handle error recovery. Generally speaking, components log their intention to perform an operation on its permanent storage before actually start the operation. The logged record is removed after the data related to the operation have all been committed to permanent storages. If errors do happen in the middle of an operation, the logged record on the permanent storage is used to replay un-committed tasks. Fortunately, our data migration tasks do not generate new data by themselves. Therefore, there will be no data lost due to memory cache.

Due to the complexity of the error recovery scheme and the common understanding of journalling mechanism, we omit the details of the logging in this paper.

5. Prototyping and Performance Evaluation

In order to prove the feasibility of our proposed scheme, we have done a prototyping on Lustre file system. In this

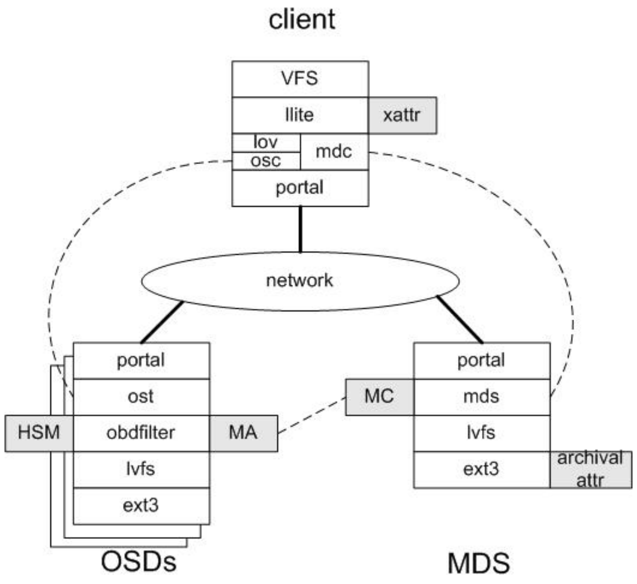


Figure 6. Lustre modules and prototyping components

section, we briefly describe our prototyping and then study its performance. We also share some experiences we learnt through our prototyping process.

5.1. Prototyping on Lustre

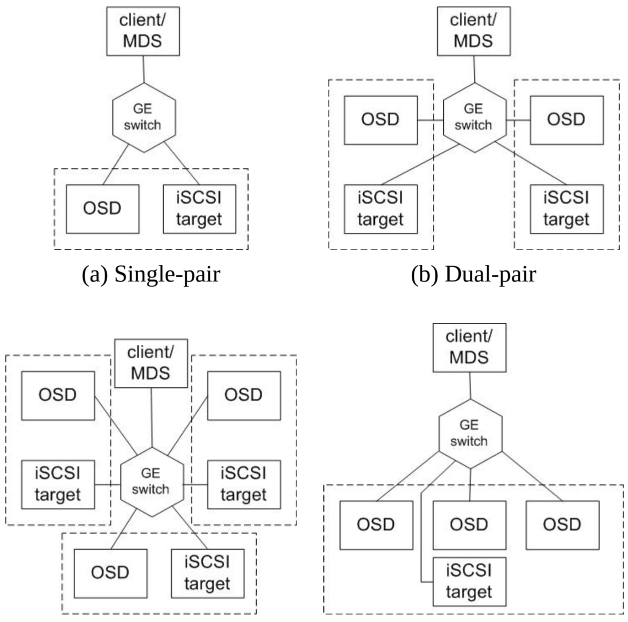
Lustre is a scalable, secure, robust, highly-available cluster file system [2] on Linux operating systems. It is open source so that it is possible for us to develop our add-on functions on top of it. Lustre has the exact architecture shown in Figure 2. Figure 6 is an anatomy of Lustre nodes into functional modules. We will not explain the detailed functions of each module in this paper but interested readers can refer to [2]. All white blocks are modules already in Lustre and gray blocks represents new components or functions developed for parallel archiving. Instead of developing separate modules for the migration agent and the migration coordinator, we decided to instrument existing Lustre modules to reuse existing codes. We implemented the migration coordinator as part of the *mds* module that handles metadata queries. In Lustre, all metadata are stored in the local ext3 file system in the MetaData Server node. The migration agent is implemented as part of the *obdfilter* module that translates object access requests to local file system access requests. The archival attributes are stored as extended attributes of the ext3 file system in the metadata server. All local ext3 file systems in Lustre nodes (MDS and OSD) are accessed through *lvfs* modules that is a Lustre-tailored Virtual File System (VFS). Another local file system currently supported by Lustre is ReiserFS. The *llite* module in client nodes are modified to query and

CPU	Two Intel XEON 2.0GHz w/ HT
Memory	256MB DDR DIMM
SCSI interface	Ultra160 SCSI (160MBps)
HDD speed	10,000 RPM
Average seek time	4.7 ms
NIC	Intel Pro/1000MF

Table 2. Configuration of OSD hosts

CPU	Four Pentium III 500MHZ
Memory	1GB EDO DIMM
SCSI interface	Ultra2/LVD SCSI (80MBps)
HDD speed	10,000 RPM
Average seek time	5.2 ms
NIC	Intel Pro/1000MF

Table 3. Configuration of iSCSI target hosts



(a) Single-pair (b) Dual-pair

(c) Triple-pair (d) Singe-backup

Figure 7. Testing configurations

lock archival attributes of files before sending out read or write requests for objects to any OSD nodes through local *osc* modules, which is represented by the *xattr* gray block in Figure 6. The *llite* module implements a light-weighted client file system. Its function is similar to the NFS client file system in NFS implementations.

Since we did not find any open source HSM software, we developed our own with restricted functions. Specifically, our HSM can only migrate files between two file systems. We have a kernel thread waken up periodically to look for files in the file system of the online storage for backup but we did not implement any sophisticated policies about when and what to backup. Our HSM codes are also part of the *obdfilter* module of Lustre. When initializing a HSM instance, a backup storage device is specified and typically mounted as an ext3 file system. Our HSM also use the *lvfs* module to access the backup file system. In our experiments, we use iSCSI protocol to access remote backup storage devices, which are exposed as an ordinary SCSI disk device on OSD nodes.

5.2. Experiment Setup

Our prototyping is on Lustre version 1.4.0 and Linux kernel 2.4.20 of RedHat 9. In all of our experiments, we run OSD on up to 3 machines with configuration described in Table 2. We use the iSCSI reference implementation developed by Intel [10] to set up our backup storage devices. Up to 3 machines run as iSCSI targets in our experiments and Table 3 contains the configuration of these target machines. When the OSD machines have loaded iSCSI initiator drivers, a new SCSI disk device is registered and become available (e.g., /dev/sdc). We then create one or more than one partitions on it and make ext3 files on each partition. When we setting up the HSM instance on the OSD, the device name of one partition like /dev/sdc1 is specified so that it becomes the backup storage device of that OSD. In all experiments, client and MetaData server are running on the same machine with the configuration similar to Table 3. All machines are connected to a Cisco Catalyst 4000 Series gigabit ethernet switch.

With our available hardware resources, we have designed four different configurations as illustrated in Figure 7. Configurations (a), (b) and (c) represents scaling up the system by adding more pairs of OSD and backup storages. Configuration (d) represents the single-backup point setup, where multiple OSDs share the same backup storage. Note that even configuration (d) has multiple parallel migration paths but they unfortunately collide at the entry point of the backup storage. In all configurations, multiple OSDs are configured as RAID 0, i.e., files are striped as multiple objects on all OSDs. In the first three configurations, we create one partition on each iSCSI ”disk”. In configuration (d), we create three partitions, each of which is used by one of the three OSDs as backup storage. We cannot let the three OSDs working on the same partition since the file system may get corrupted.

Our prototyping allows us to manually trigger the migration of objects between OSDs and backup storages. When data are backed up or recalled, we measure the throughput achieved on each pair of OSD and backup storage. We also measured the latency between the moment when the migration agent initiates the parallel backup/recall operations and the moment that all replies are received. Finally, we measure the latency perceived by applications on client hosts when accessing files on backup storage. We use a simple program reading the first byte of a specified file. When the

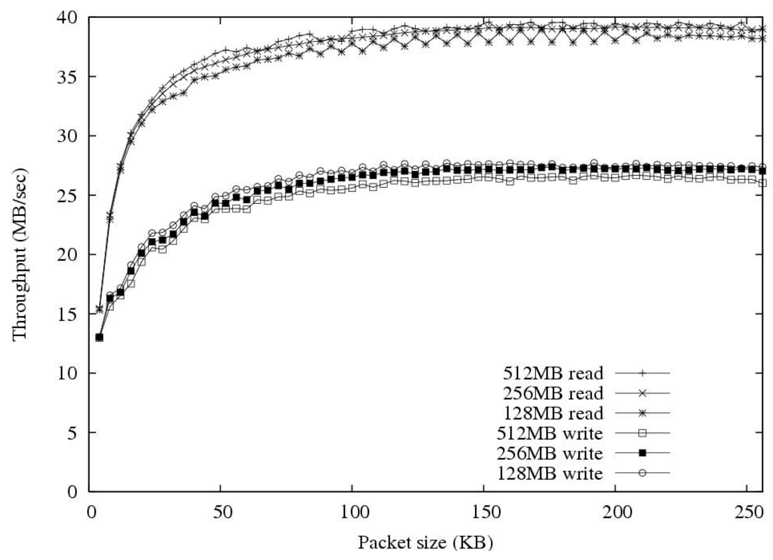


Figure 8. Throughput of iSCSI

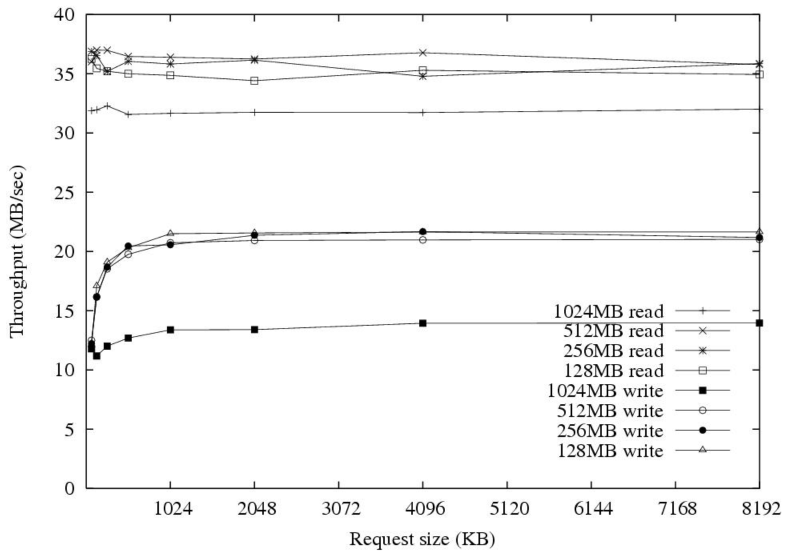


Figure 9. Throughput of ext3 file system over iSCSI

file is on backup storage, the system automatically migrates all the objects of the file back to OSDs before serving the reading request.

5.3. Performance Results

Before we start to test our prototyping system, we want to know how much throughput we can achieve between OSD machines and iSCSI target machines using iSCSI as a baseline. Figure 8 illustrates such achievable throughput. We have 3 pairs of OSD and iSCSI target machines so the numbers are averaged across them. For each pair, tests are performed 10 times and the average numbers are reported. The iSCSI code has a packet-size limit of 256KB so we test the packet size from 4KB up to 256KB with increase of 4KB at each step. All tests are sequential read or write starting from logical block address (LBA) 0 up to the test size. Figure 8 shows 3 test sizes: 128MB, 256MB and 512MB. In summary, higher throughput can be achieved for reads than for writes. Larger packet size leads to higher throughput since overheads associated with each packet are fixed. Larger data size has better read throughput. However, less throughput can be achieved on larger data size due to the high cache pressure when dirty pages need to be flushed into persistent storage devices of iSCSI targets.

In the previous test, the throughput is measured for raw iSCSI requests. Since the HSM instances in OSDs mount the iSCSI devices as ext3 file systems and perform file system operations when migrating files, we use *iozone* to measure the file system throughput shown in Figure 9. These numbers are therefore the approximate maximum throughput we can achieve on one pair of OSD and iSCSI target machines. In the tests, we let iozone perform synchronous IO when writing since we want to disable the delay-writes because of the OSD cache and get the real throughput to the iSCSI target instead of OSD memory. In our prototyping, files on file systems of backup storage devices are always open with *O_SYNC* flag to force synchronous writes. In this set of tests, we test request sizes beyond 256KB. However, we see little increase in throughput since the underlying iSCSI transport always breaks requests to a maximum packet size of 256KB. For the file size of 1GB, we see a substantial drop in the throughput in both read and write. This is because that the iSCSI target machines have memory capacity of 1GB and the slower disk access starts to take over for large file sizes.

We demonstrate the scalability of our proposed architecture by measuring the aggregated throughput achieved in the four configurations described in the previous section. From the previous two sets of tests, we know that packet size of 256KB achieves nearly the maximum throughput so we use this packet size in the all tests. We run each combination of configuration and file size five times and report the average numbers in the following.

Figure 10 shows the achieved aggregated backup throughput of each configuration over four different file sizes. The throughput achieved on each pair of OSD and iSCSI target is also show as sections of the bars. By looking at the single-pair, dual-pair and triple-pair configurations, we can see the linear increases in backup throughput with the number of pairs. Comparison of the triple-pair and single-backup configuration demonstrate that single backup point can not scale up very well. In all configurations, the aggregated throughput decrease as the file system increases due to the increase memory pressure in the iSCSI target. However, the single-backup configuration drops much faster than the dual-pair and triple-pair configurations since the other two also scale up their memory when adding new pairs.

Figure 11 shows the aggregated recall throughput achieved in the four configurations. The high throughput achieved in file sizes of 128MB and 256MB are due to the caching effect in OSD. We measure the recall throughput by first migrate the file objects from OSDs to iSCSI targets and then immediately recall them so that part of the files blocks may already in OSDs' buffer cache. We configure the OSD hosts with only 256MB memory so the raw iSCSI throughput starts to take over in file sizes over 256MB. In file sizes of 512MB and 1GB, we can again observe the near linear scalability in recall throughput with increasing number of pairs of OSD and iSCSI target.

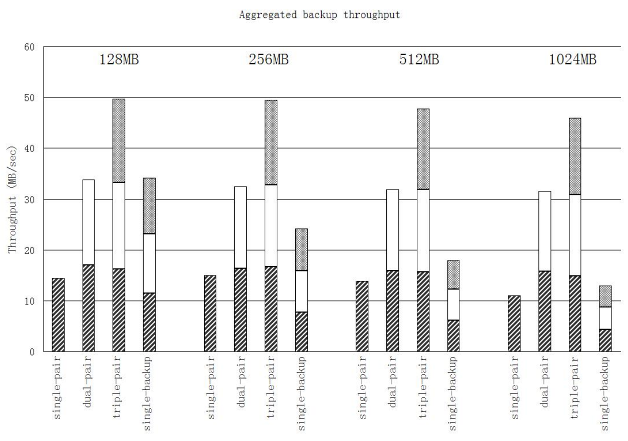


Figure 10. Aggregated backup throughput

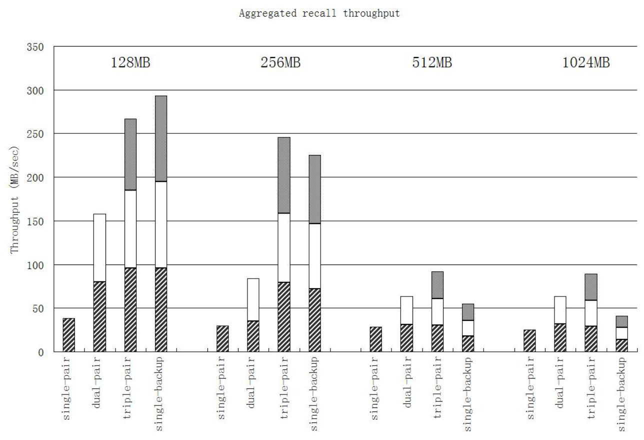


Figure 11. Aggregated recall throughput

The latency perceived by application when accessing a file whose objects are stored on backup storages is illustrated in Fiugre 12. We see that the improvement from the dual-pair configuration to triple-pair configuration is not as dramatic as from the sing-pair configuration to the dual-pair configuration. This is due to the fixed overhead spent in places other than IO. The singe-backup configuration again shows poor scalability compared with the dual-pair and triple-pair configurations.

5.4. Experiences in Prototyping

Our proposed scheme requires the MDS to answer queries of file layout information given input parameters of OSD identification and object identification. The Lustre MDS uses an ext3 file system to manage the namespace and stores file layout information as extended attributes of files. It can not answer the aforementioned query efficiently. In our prototyping, we take a brutal force hacking to traverse the entire file system tree and extract layout attribute of

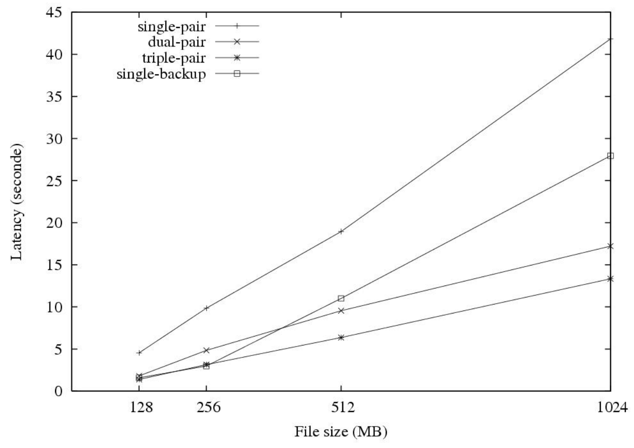


Figure 12. Application perceived recall latency

each regular file to check whether the requested object is part of it. A more elegant solution is definitely needed for file systems with thousands or millions of files. One potential approach is to create a special directory grouped according to OSD identification number and object identification numbers so that a pair of them can be directly translated into a pathname of a regular file. This regular file is a symbolic-link or hard-link to the real file or file inode containing the object. Another approach may be using a separate database to manage this mapping. However, both approaches requires extra efforts to keep the consistency of mappings.

Another problem we encountered is caused by the timeout mechanism used by Lustre to discover component failures. For our user program to test recall latency, we have to increase the system timeout parameter so that recall of large files can finish in time. This actually is just one of many problems appear when we are moving from single machine systems to multiple-component cluster systems. New user interfaces need to be designed. In our case, interfaces that specific to files potentially on backup storage may be needed.

6. Related Work

Object-base storage is an emerging standard designed to overcome the functional limitations in current storage interfaces (SCSI and ATA/IDE). The Network Attached Secure Disks (NASD) project [11] at CMU is a pilot study in object-based storage. This work led to a larger industry-sponsored project under the auspices of the National Storage Industry Consortium (NSIC) generating a standard extension to the SCSI protocol for object-based storage [12]. The Storage Networking Industry Association (SNIA) continues to define the NSIC draft and submitted a completed SCSI draft standard to T10 in 2004 [5]. Although the standard is still under developing, the industry is already implementing systems using the object-based storage technologies. Examples include the IBM's next generation StorageTank [13], the highly scalable Lustre file system [2] by Cluster File Systems Inc., ActiveScale storage clusters [3] from Panasas and so on.

There are several systems based on block devices providing transparent HSM functions. VERITAS' NetBackup Storage Migrator [14] is a classic DMAPI-based HSM solution for single server file systems. It has implementations on versatile file systems including OnlineJFS, XFS and VxFS. SGI InfiniteStorage Data Migration Facility (DMF) [8] is another single server HSM solution primarily on the SGI XFS file system. DMF can also be combined with SGI's cluster file system CXFS [15]. However, DMF is required to be installed on the host working as the metadata server of CXFS so the architecture is still similar to that of single server file systems. IBM's High Performance Storage Systems (HPSS) [9] is a cluster storage system on top of IBM General Parallel File System [16]. Its HSM function also relies on the DMAPI support of GPFS. However, XDSM/DMAPI was originally designed for single server file systems. GPFS has an extension of XDSM/DMAPI for a cluster environment. A dedicated server is setup to receive data management events generated in client hosts. This server thus performs the core function of hierarchical storage management and is call the *Backup Core Server*. Since dumb block devices cannot copy data to each other directly, special hosts called *Tape-Disk Movers* are attached to SAN to perform data migrations under the control of the *Backup Core Server*.

7. Conclusions

In this paper, we have proposed a parallel hierarchial storage architecture in object-based cluster file systems. Our main contribution is a coordinating scheme to fully explore the parallel migration data paths in this architecture. We have developed a prototyping system on Lustre file system as a proof-of-concept. Through performance studies of this prototype, we have demonstrated the scalability of our scheme.

References

[1] SGS file system RFP. Technical report, DOE NNCA and DOD NSA, April 25 2001.

[2] Lustre: A scalable, high-performance file system. Whitepaper, Cluster File System, Inc. http://www.lustre.org/docs/lustre.pdf.

[3] Panasas. Activescale file system. http://www.panasas.com/panfs.html.

[4] Gary Grider. Scalable i/o, file systems, and storage netwoks:R&D at Los Alamos, May 2005.

[5] SNIA. SCSI object-based storage device commands (OSD). T10 working draft. http://www.snia.org/osd.

[6] M. Mesnier, G. Ganger, and E. Riedel. Object-based storage. *IEEE Communications Magazine*, 41(8):84–90, August 2003.

[7] The Open Group. System management: Data storage management (XDSM) api. Technical standards, Jan 1997. ISBN 1-85912-190-X.

[8] Laura Shepard. SGI infinitestorage data migration facility (DMF) a new frontier in data lifecycle management. White paper, SGI. http://www.sgi.com/pdfs/3631.pdf.

[9] Richard W. Watson. High performance storage system scalability: Architecture, implementation and experience. In *Proceeding of 22nd IEEE - 13th NASA Goddard Conference on Mass Storage Systems and Technologies*, 2005.

[10] IETF. Internet small computer system interface (iscsi). Rfc 3720. http://www.ietf.org/rfc/rfc3720.txt.

[11] Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A cost-effective, high-bandwidth storage architecture. *SIGPLAN Not.*, 33(11):92–103, 1998.

[12] R. Weber. Object-based storage devices (osd). http://www.t10.org.

[13] IBM Almaden Research. Storage tank. http://www.almaden.ibm.com/StorageSystems.

[14] Veritas netbackup storage migrator for unix. White paper, VERITAS Software Corporation.

[15] Laura Shepard and Eric Eppe. SGI infinitestorage shared filesystem cxfs: A high-performance, multi-os filesystem from sgi. White paper, SGI. http://www.sgi.com/pdfs/2691.pdf.

[16] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proc. of the First Conference on File and Storage Technologies (FAST)*, pages 231–244, January 2002.