Design and Implementation of a Network Aware Object-based Tape Device

Dingshan He, Nagapramod Mandagere, David Du
DTC Intelligent Storage Consortium
University of Minnesota, Twin Cities
he@cs.umn.edu,npramod@cs.umn.edu,du@cs.umn.edu

Abstract

Data storage requirements have constantly evolved over time. In recent times, the rate of increase in volume of data has been exponential, partly because of regulations and partly because of increase in richness of data. This trend has led to an equally explosive increase in cost of management. Intelligent Storage Devices built using Object-based Storage (OSD) Interfaces have gained increased acceptance due to the benefits of reduced management costs.

The command set of the current OSD standard does not work well with tape-based storage solutions. In this work we propose a few extensions to the OSD standard in order to facilitate easier integration of tape devices into the object storage ecosystem. Further, we propose an intelligent buffering mechanism to maximize the utility of network attached tape devices, and we also propose mechanisms to make tape cartridges more portable within any object storage ecosystem.

Our results for the intelligent buffering mechanism show that our scheme adapts better to mismatches in network and tape drive bandwidths. Specifically, our scheme helps minimize the repositioning of the tape, leading to better utilization and increased lifetime of the tape.

1. Introduction

An explosion in the volume of data storage primarily triggered by digitization of tons of non-electronic data, the growth of the Internet as a platform (Web 2.0), and several other factors have put extreme demands on current storage solutions. Both the performance and the financial costs of storage systems have become causes of extreme concern. The total cost of ownership, which includes initial acquisition cost and maintenance or storage administration costs, has seen an unprecedented increase, the majority of which is attributed to administration costs. In order to minimize these management costs, we need to make storage devices more intelligent, self-managing, self-tuning, and adaptive.

Intelligence in general requires inherent, in-depth knowledge of the metadata, which is lacking in current systems. Block-based interfaces do not allow for communication of metadata between the applications that use the storage and the storage devices that end up housing the data. These limitations have led the industry and academia to rethink the way data is stored. The NASD project at CMU, [1] and [2], pushed forward the idea of storing data as objects. Conceptually, an object is a wrapper around a block which stores relevant metadata. An interface to communicate this metadata is one of the most important aspects of this model. Lustre file system [3] and the Panasas ActiveScale file system are two popular examples. Both of these systems use proprietary object storage interfaces, which leads to concerns regarding interoperability. These concerns, along with acceptance of the importance of object-based storage interfaces, led to standardization efforts. In late 2004, the first version of the T10 OSD (Object-based Storage Device) standard was released, and work continues on drafts of the second version of the standard [4].

The OSD standard is designed with the implicit assumption that the storage device is a random access device, like disk, but in an object-based storage ecosystem there is still need for tape devices to satisfy archival and backup requirements. Tape storage is still the most cost effective and reliable way to archive and back up data for long periods of time. Close to 90% of enterprises use some kind of tape-based archive and backup scheme [5]. This continued popularity of tape could be attributed to several factors, chief among them are the cost and capacity of tapes. The capacity of tape storage is doubling every two years [5]. A T10000 tape cartridge from Sun/StorageTek has a raw capacity of close to half a terabyte. With compression, even higher capacities can be achieved. There has been some shift towards use of disk-based archive and backup systems due to the fact that disks have also grown in capacity and have become cheaper, but tapes still hold the edge due to their portability and energy efficiency. Tape cartridges can be shipped offsite for vaulting purposes, and when they are not being accessed they consume no energy.

Use of traditional tape devices with block or NAS interfaces in object storage ecosystems has several limitations. A block interface is complicated to manage and difficult to share among multiple higher-level object-based storage devices. It requires us to run a dedicated file system instance on the source OSD to manage the data stored. If the archiving device is a tape device exposing its native SCSI Streaming Command (SSC) interface, the source OSD then needs to have corresponding tape management software. In addition, multiple higher-level OSD devices cannot correctly share the same tape device using a block or streaming interface if they do not coordinate with each other. A NAS interface is incompatible with the object interface and lacks efficient support for object attributes. Using a NAS device as the archiving device also requires the higher-level OSD devices to run additional NAS clients. More important, NAS runs security methods that are incompatible with OSD, so as objects are archived to NAS devices, there is no longer an integrity guarantee for the objects. Any NAS clients can access and modify the archived data without going through the security checking of the OSD model. Another limitation of the NAS model relates to support of attributes. Object attributes are essential information to describe the features of objects and are helpful in searching for them. Therefore, object attributes should always go along with the object data into archival storage. A NAS storage device relies on its local file system to provide support for file attributes.

The focus of our research is to investigate the possible extensions to the OSD T10 interface to support tape devices and to identify and propose solutions to make integration of tape devices into an object storage ecosystem simple and efficient. Some of the challenges that we have identified include,

Challenge 1: *Extensions to OSD Standard*
Many of the existing data manipulation commands in existing object storage interfaces are neither suitable for the sequential-access characteristic of tape devices, nor necessary for the dedicated application of our Object Archive Tape Device, i.e., archive/restore. For example, SNIA OSD and Lustre OST both have CREATE, READ, and WRITE commands to allow clients to create an object and write data into or read data from the object at any offset of the object at any time. If such commands are issued to the Object Archive Tape Devices, the result would be frustrating at best. The random small read commands will cause frequent slow repositioning of the tape while the write commands, in addition to positioning tape, may cause data loss if it is over-writing existing data instead of appending at the end of data. The data loss is due to a special feature of tape drives where every write command puts an end of data (EOD) mark on the tape following the written data, and any data past the EOD mark on tapes becomes inaccessible.

High-level object storage devices directly communicate with the Object Archive Tape Device to archive or restore objects. Since one Object Archive Tape Device can be shared by multiple high-level object storage devices, the archive/restore operations cannot be a simple sequence of object WRITE/READ commands, which could cause competition for the limited number of tape drives. If we make tape device implement and expose the OSD T10 interface directly, movement of objects between disks and tapes would be simple migration tasks. However, the limitation here is that a tape device could be shared amongst multiple devices, and hence, if each device does random requests to the tape, performance would degrade.

In addition, object attributes should also be archived/restored as part of the object. Therefore, in order to archive an object from a high-level object storage device to an Object Archive Tape Device, the CREATE command, SET ATTRIBUTE commands, and WRITE commands should be executed as one transaction. Tape drives are assigned to one archive session exclusively so that object attribute lists and data are stored continuously on the tape. Similarly, the sequence of commands to restore objects from an Object Archive Tape Device to a high-level object device should also be executed as one transaction and use the assigned tape drive exclusively to maximize retrieval speed. The key challenge here is to design an efficient means to ensure transactional control at the level of archive and restore operations.

Challenge 2: *Addressing the Bandwidth Mismatch*
In the object-based storage ecosystem, the number and type of OSDs can vary greatly across deployment. At one end, we have each disk exposing an OSD interface, in which case the ecosystem could be made up of millions of OSD devices. At the other end, we have disk arrays exposing an OSD interface, in which case the number of OSDs is relatively smaller. In either case it is highly likely that multiple OSDs share a tape library due to data consolidation needs. In such cases, to ease the process of sharing and for availability reasons, a tape library should facilitate direct connection to a network. The main problem with this approach is that this creates a mismatch between the network bandwidth and the tape drive bandwidth. Also, advanced disk subsystems typically use RAID technology to multiply their data transfer rates so that they may supply data at a rate faster than a tape drive. When the data transfer rate of an archive/restore session is faster than that of the tape drive, the tape data transfer rate becomes the bottleneck. On the other hand, building RAID over tape drives [6][7] has been shown to be detrimental in the presence of concurrent users due to the small number of drives in tertiary libraries [8].

These mismatches lead to a problem called tape itch, or tape repositions. Tapes work best in streaming mode. If a host can supply data at a rate greater than or equal to the bandwidth of the tape drive, maximum utilization can be achieved. However, if a host cannot supply data at a rate greater than tape bandwidth, data underrun occurs and the tape drive has to reposition. Once the drive begins to reposition the tape, the drive controller can receive data from the host into its buffer, but it cannot supply data to the tape drive until the drive completes its reposition. During reading, repositioning can also occur if the host system cannot extract data from the controller buffer quickly enough. This reposition time may vary from several milliseconds to a few seconds depending on the technology of the drive. Frequent repositioning can significantly slow the overall performance of a tape drive from its rated speed. Moreover, repositioning can cause wear to the tape media and drive head, thus affecting the lifetime of tapes and tape drives.

Challenge 3: *Making Tapes Truly Portable*

In enterprise storage solutions, due to the enormous volume of data, often data is moved from one place to the other using various HSM (Hierarchical Storage Management) solutions. In the long run data archived from one storage device might need to be restored onto a totally different storage device. In such scenarios, the tape cartridges used to archive data need to be autonomous. Specifically, if all the necessary metadata needed to interpret the data is not available on the tape, the tape cartridge could prove to be useless. In addition, if one inserts a tape cartridge containing archival data from some other storage device into an existing system, we need mechanisms by which this new tape metadata can be combined with the global metadata of the overall system. This is especially important from the point of indexing and lookup operations that are used to catalog contents of the system for efficient access. In summary, the main challenge here is the mechanism of making tape cartridges autonomous or self-identifying.

In this work we first propose extensions to the OSD T10 Standard to efficiently support archive and restore operations. Next, we propose an intelligent disk based buffering scheme to help minimize the impact of mismatch in bandwidths on the performance of archival solutions. Last, we propose mechanisms to make tape cartridges more autonomous in an object storage ecosystem. The rest of the paper is organized as follows: Section 3 discusses the limitations of the current tape libraries. Section 4 highlights our proposed extensions to enable efficient archival operations. Section 5 presents our proposed intelligent buffering scheme. Section 6 describes the proposed mechanism for self-identifying tapes. Section 7 describes the performance results, and our conclusions are presented in Section 8.

2. Related Work

One of the major limitations of tapes is their access latency. While the seek time of disks is around 3-10 milliseconds, the average tape seek time can reach over a minute depending on the length of the tape. If the requested tape is not already mounted on a tape drive, additional switch time of tens of seconds is needed. This feature limits the effective use of tape devices to sequential applications including archiving. Any attempt to use tapes in a random access way will result in frustratingly slow performance. There have been many studies on how to align objects on the tape media to minimize the access latency [9][10]. In addition, how to schedule access requests in order to optimize the usage of sequential-access tapes and the limited number of tape drives is also covered by previous research [10][11][12][13]. Our OATD (Object Archive Tape Device) adopts some of these schemes for optimizing access latency, so these issues are not our research focuses in this study. There are two existing mechanisms often adopted to compensate for the mismatch of data transfer rates between the network and the tape drives. The first one is interleaving (also known as multiplexing or multi-streaming), which combines the streams of data from multiple sources into the single data stream written to the tape drive [14]. While this approach is successful for improving archive performance in cases with a sufficient number of concurrent slow archive sessions, the potential failures and overhead for restore operations become a major concern. Depending on the amount of data and number of archive sessions interleaved into a single stream, restoring the data for a single object could require reading much more data. For archive sessions with a faster network data transfer rate than the tape drive, interleaving with other concurrent sessions is not beneficial at all. The second approach is disk staging. When archiving objects, the actual data streams are written to a disk volume instead of tape. Then, the staged archives are copied onto tape media. This mechanism is not adaptive, hence it will stage data in cases where there is no significant mismatch between bandwidths, leading to sub-optimal performance.

3. Limitations of Current Tape Libraries

The main problem in attaching a tape device directly to the network is that the network bandwidth and tape drive bandwidth often do not match. The network bandwidth (B_{net}) dictates the rate at which data is received by the tape device and the rate at which it is available for writing to tape. The tape drive bandwidth (B_{tape}) dictates the rate at which data can be written onto a tape cartridge. Figures 1, 2 and 3 illustrate the problem of mismatch between B_{net} and B_{tape} for an *archive* operation. A similar effect can be seen with restore operations. The slope of the line indicates the bandwidth. For performance reasons, usually a fixed-size double buffer is used as a level of indirection between the network interface and the tape drive. The incoming data is written to buffer B_1 and once it is full, the tape starts writing this buffer to the tape cartridge. In the meantime all the new incoming data is stored in buffer B_2. Once the tape drive finishes writing B_1 to tape, it checks to see if buffer

B_2 is ready/full. If not, the tape drive goes into repositioning in order to ensure contiguity of written data. Thus the tape and network toggle between the two buffers. Based on the values of B_{tape} and B_{net}, multiple scenarios come to light. Figure 1 shows the case with $B_{tape} > B_{net}$. The data

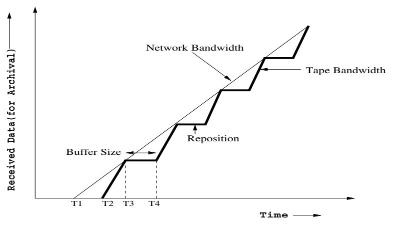


Figure 1. $B_{tape} > B_{net}$

for an archive session starts arriving at time T_1. The first buffer unit is filled and sent to tape for writing at time T_2. Here, the time taken by the tape drive to write the buffer unit to tape is less than the time taken by arriving data to fill up the next buffer unit. Hence, the tape drive is forced to reposition at time T_3. Once a tape drive starts repositioning, even if data becomes available, the system has to wait for a certain amount of time. This tape drive reposition time differs across products, but it is usually on the order of a few milliseconds. At time T_4, the data buffer becomes available and, assuming that repositioning is complete, the tape can again start writing data.

Figure 2 shows the case with $B_{tape} < B_{net}$. Here, an infinite buffer is assumed. At time T_1 data for an archive session starts arriving. At time T_2, the buffer unit is full and the tape drive starts writing this data into the tape cartridge. Since the tape drive does not finish writing the buffer before the arriving data fills up the other buffer, the buffer size continues to grow with new incoming data. Based on the difference in B_{net} and B_{tape}, the maximum buffer size (approximate) required to buffer all incoming data is given by the relation,

$$Buffer\_Size_{max} = Sizeof(object) * \frac{B_{net} - B_{tape}}{B_{net}} \quad (1)$$

Based on the size of object and extent of the disparity

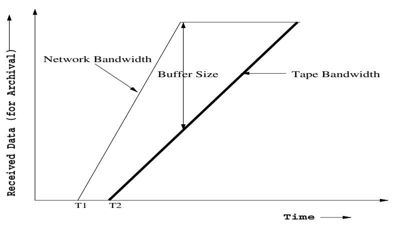


Figure 2. $B_{tape} < B_{net}$ **with unlimited buffer**

between the bandwidths, the amount of memory required varies, and the worst case space requirement is not bound.

Figure 3 shows the case $B_{tape} < B_{net}$, but with limited buffering. The buffer size is fixed and a feedback is created between buffer usage and data arrival rate. If the buffer is full, the archival application is asked to slow down. This approach is very common but has several limitations. First, the overhead for communicating the buffer usage information is high. Second, this leads to a longer archival time perceived by the initiating storage device.

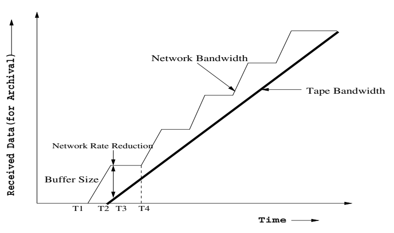


Figure 3. $B_{tape} < B_{net}$ **with limited buffer**

In order to mitigate the effects of mismatch mentioned above, two techniques are commonly used, namely *Interleaving* and *Disk Staging*. *Interleaving* involves multiplexing or combining multiple streams of data from different sources into a single stream and writing it to tape [14]. When a large number of slow concurrent archival sessions exist in the system, interleaving helps achieve good archival performance. However, due interleaving, any session's data is no longer contiguous on tape. Hence, restoring a single object might necessitate reading a lot more unwanted data which adversely impacts restore times.

Disk Staging involves reading or writing from/to disk volumes in front of the tape drive. The staged data from these disk volumes is later copied to tape for archiving or over the network for restore. This is a very static scheme and does not perform well across different workloads. If the large number of concurrent slow sessions is in progress, disk staging can give a substantial increase in performance. However, if the disparity between tape and network bandwidth is very small, disk staging adds an unwanted penalty to the system. In essence, it amounts to performing two archive/restore calls for each call, leading to longer response times.

4. Design of Archive/Restore Functionality

Figure 4 depicts the system environment of an object-based cluster file system that uses an Object Archive Tape Device as archiving storage. The Object Archive Tape Device exposes a high-level object interface to the outside and performs the space management internally. However, both the object interface and the space management of the Object Archive Tape Devices are different from any existing object storage interface, which are all based on disk storage.

This object interface is provided by a front-end host that we have implemented in our design. We assume this

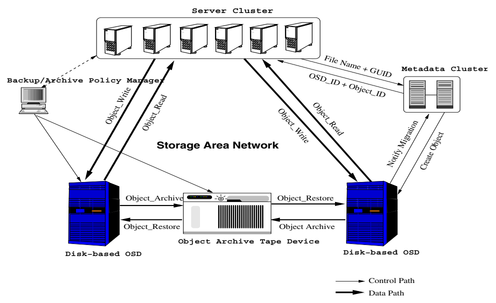


Figure 4. Object Storage Ecosystem

front-end host has multiple disk drives to be used as disk buffers and to store management information, which will be elaborated in following sections. There are one or more tape drives attached to the front-end host. The communication protocol between the front-end host and the tape drives is the SCSI Stream Command (SSC) set. It is the front-end host that translates object commands to streaming commands. In addition, the front-end host also controls the auto-loader, which uses the SCSI Media Changer Command (SMC) set to transfer tapes between tape drives and the cells holding inactive tapes. Considering the size of a typical tape library, it is reasonable to assume that the front-end host can be built into the library.

The ARCHIVE and RESTORE commands are initiated by some archive/backup application sitting inside an object-based storage target. This in effect acts like a distributed Hierarchical Storage Manager which sends ARCHIVE or RESTORE commands to the Object Archive Tape Devices. It is also quite possible that the archive/backup application is running on a node other than the object storage device. In this case, the node will send a simple control message to the higher-level object storage device to initiate the archive or restore operation. This ARCHIVE/RESTORE interface of object storage greatly simplifies the archive/backup applications. This simplification helps such applications only focus on policy definition.

Both ARCHIVE and RESTORE protocols involve multiple rounds of message exchange between the higher-level object device and the Object Archive Tape Device. We design the protocol based on the T10 OSD standard and extend the message format defined in the standard [4]. In the following description of the protocols, we only point out the extensions we propose. For the parts that are inherited from the T10 OSD standard, the reader should refer to the standard document [4] for explanations.

4.1. ARCHIVE Protocol

For the purpose of archival operation, the higher-level disk-based OSD acts as the initiator and the object-based tape device acts as the target. Any client or dedicated archival client can issue an archive command to the source disk-based OSD. The arguments for this command are the partition and object identifiers of the user object. On receiving an archival request from any client, the disk-based OSD performs the following actions:

- Lock the user object for the purpose of concurrency control.
- Retrieve all attributes that belong to the user object and format them into an attribute list.
- Construct and send to OSD tape device an OSD ARCHIVE CDB (Command Descriptor Block) with the source object identification, length of attribute list, and length of data object. (No actual data is sent in this step.)

Upon receiving an OSD ARCHIVE command, the tape device first checks to see if enough space is available on the mounted tape cartridge. If not, it unloads the cartridge, loads an empty cartridge, and locks the tape to prevent other sessions from accessing the tape at the same time. Once ready with a new or empty tape, the tape device performs the following tasks:

- Assign a Task Tag for this request and create data structures to record parameters like attribute list length, data length, and Task Tag.
- Create a response message with the Task Tag and send it to the OSD disk device. To uniquely identify the task, this same Task Tag is used for all future data packets sent by the source that belong to this object.

After the source OSD receives the reply message of the OSD ARCHIVE command, it starts a multi-round data transfer process to send the attribute list plus data to the Object Archive Tape Device. For each round, an OSD ARCHIVE APPEND command is used to transfer a block of the attribute list or data.

The sender Offset parameter indicates the offset of the first byte of the current block in the entire object byte stream: attribute list concatenated with data. The Task Tag is the one received from the reply message of the OSD ARCHIVE command. When receiving such an OSD ARCHIVE APPEND command, the Object Archive Tape Device uses the Task Tag parameter to locate the archive task. The content of the Data-in buffer is copied to the task buffer queue for dumping to the tape asynchronously. A counter in the task data structure used to keep track of the number of received bytes, recvOffset, is updated. The tape device replies with the recvOffset and the taskTag. The recvOffset and the sendOffset are used together as a sequencing mechanism to verify that no message is missing.

The last round of the OSD ARCHIVE APPEND command causes the Object Archive Tape Device to flush all buffered data to the tape, update the tape metadata to indicate a new object, and update the tape library catalog information so the object can be located in the future. After this step, the archive task at Object Archive Tape Device is done, the task can be destroyed, and its associated resource can be released, including unlocking the tape drive so that other requests can proceed.

After the source OSD receives the reply for the last OSD ARCHIVE APPEND command, it can also reply to the archive application that initiated this entire archive process. The address of the Object Archive Tape Device is contained in the reply message. It is up to the archive application as to how to use this information. Typically, these numbers will be recorded somewhere with a mapping to the source object identifier. With such mapping information, when there are requests for the source object, a restore process can be automatically initiated. An easy way of achieving this mapping or indirection is by causing the successful completion of archive operation to update a metadata attribute of the object in the MDS to indicate the position in the storage hierarchy. This would in effect achieve DMAPI like functionality without much overhead.

4.2. RESTORE Protocol

A restore operation is usually invoked when a client tries to access an object that is in archival storage. This causes a trap to the archival application, which uses the partition and user object information to map to the corresponding object identifier on archival storage. The archival application passes this information to the disk-based OSD. On receiving such a notification from the archival application, the disk-based OSD performs the following tasks:

- Create an empty object with the object identifier provided by the archival application. This object is created in the partition indicated by the application object. It also creates a temporary container to receive the attribute list.
- Construct an OSD RESTORE command with the object identifier and partition information, and dispatch it to the tape device.

Upon receiving an OSD RESTORE command, the object tape device performs the following tasks:

- Using the object identifier, locate the tape cartridge using the index structure, which is maintained in fast-access primary memory.
- After locating the tape cartridge, load the the cartridge into any available tape drive and lock it to ensure exclusive access.
- A unique Task Tag is created just as in the archival case, but the metadata now comes from the object metadata on the tape cartridge itself. Record the sending offset, and reply to the disk-based OSD with the Task Tag, attribute list length, and data length.

Upon receiving the response for the OSD RESTORE command, the disk OSD starts a multi-round data transfer process using the OSD RESTORE READ command. Again, as in the archival process, the send offset and receive offset are used as a sequencing mechanism. The last round of the OSD RESTORE READ command causes the tape device to destroy the restore task and release its corresponding resources, which, among other things, involves releasing the lock on the tape cartridge. The disk OSD, on receiving the data from the last OSD RESTORE READ command, formats the attributes accordingly and sends a notification to the archival application that initiated the restore operation.

4.3. Error Recovery

The data transfer protocols described above, if implemented by themselves, are very susceptible to error conditions. Errors could happen during the execution of an archive/restore process due to several factors; chief among them are failure of disk-based OSD, failure of tape device, and network failure. Any error recovery mechanism to address these failure conditions has to satisfy the following requirements: *Consistency* – metadata should be consistent with the data. If an object is only partially archived or restored before it is interrupted due to some failure, the metadata should also reflect this failure. *Checkpointing* – if any object restore/archive operation is interrupted due to some failure, the system should restart the operation from the nearest completed checkpoint and not from the beginning of the transfer.

For the archival process, we propose using the OSD ARCHIVE APPEND command with certain special arguments as a checkpoint command (length field set to all $1's$). The periodicity of checkpoints is currently hard coded. Upon receiving such a checkpoint packet, the tape device flushes all data/attribute information for that particular Task Tag to tape and sends an acknowledgment with the last flushed offset to the source. If an error occurs, the source can restart the archival process from the offset in the last acknowledged checkpoint instead of doing a complete restart. Along similar lines, the OSD RESTORE READ command can be used for checkpointing restore operations.

In summary, the proposed additional commands provide several advantages, namely - Fine grained control over migration task (at an object level), concurrency control, transactional behavior and improvement in tape drive utilization by avoiding random seek behavior.

5. Network Adaptive Tape Device

5.1. Pipelined Archive/Restore

In our design of the object tape device, we do not make any assumptions about network and tape drive bandwidths. The goals of our approach are to keep the tape in streaming mode as much as possible, to minimize the number of repositions, to maximize utilization of the tape drives, to minimize response time for archive/restore operations and to maintain contiguity of data (no interleaving).

Our proposed system uses a combination of main memory buffer and disk-based buffer placed in the front-end of the tape device. The adaptiveness comes from the fact that the tape device monitors the network data rate for each archive/restore session and determines dynamically if a session's data flows directly between the network interface and the tape drive through the main memory buffer or gets temporarily staged in the disk buffer in the front-end of the tape device. Our approach differs fundamentally from disk staging systems as our system does not store and forward objects irrespective of whether disparity exists in the bandwidths. Our approach dynamically adapts to changes in network conditions by selectively using disk buffer in combination with main memory buffers. In our scheme, some, or even all, of the data may never go through the disk, i.e., disk buffers are not always used. If the size of the requested object is small, or the difference between the session network bandwidth and the tape drive bandwidth is not too large, it is likely that the session outstanding main memory buffer does not ever reach the buffer limit, which we set to 40MB in our experiments. Also, our disk buffer is designed to work in parallel or pipelined mode supporting multiple concurrent archive/restore sessions, whereas disk staging usually works in sequential mode, i.e., first stage the entire object in a disk volume and then forward it to either tape (archive) or the network interface (restore) in a session by session fashion.

We propose to use the disk in the front-end host of the OATD as an extension of the main memory buffer only when necessary. Each archive/restore session has an upper limit on the main memory buffer that can be assigned to it. Initially, the data is buffered in main memory and at the same time consumed by either being written onto tape media for archiving or being sent over the network through the object interface for restore.

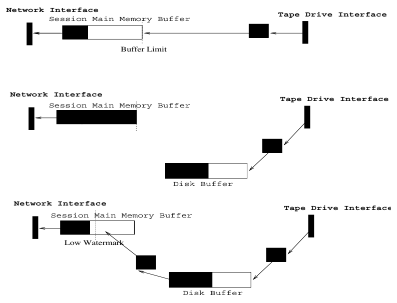


Figure 5. Pipelined Restore with $B_{tape} > B_{net}$

For an archive session with $B_{net} > B_{tape}$, or for a restore session with $B_{tape} > B_{net}$, a session could reach the limit of its main memory buffer depending on the size of the requested object and the difference between the network bandwidth and the tape drive bandwidth. When this happens, new data starts to be buffered into disk instead of main memory. At the same time, data in the session's main memory buffer is consumed. When the data left in the session's main memory buffer decreases to a low watermark, more data is moved from the disk buffer to the main memory until either no data is left in the disk buffer or the limit of the main memory buffer is reached again. Figure 5 shows the three stages in a pipelined restore session with $B_{tape} > B_{net}$. By switching the role of tape drive and network interface in the figure, similar stages of an archive session with $B_{tape} < B_{net}$ can be visualized. In order for the disk buffer not to slow down either the tape or the network, the disk data transfer rate should be larger than the sum of the session's network bandwidth and the tape drive bandwidth. With fast storage networks and fast tape drives, multiple disks working as RAID-0 may be needed. The session's network bandwidth B_{net} is specified as a QoS (Quality of Service) parameter when the session is created. The tape drive bandwidth B_{tape} is measured offline and provided to the scheduler since it is a static parameter. Depending on the session's B_{net} and B_{tape}, the type of the session, and the availability of the tape drive, the scheduler decides how to proceed as follows: For an Archival Session, if

- $B_{tape} < B_{net}$ and Tape Drive Available: Pipelined Archive.
- $B_{tape} > B_{net}$: Stage entire object to disk and copy onto tape when it becomes available.
- Tape Drive Unavailable: Stage entire object to disk and copy onto tape when it becomes available.

For a Restore Session, if

- $B_{tape} > B_{net}$: Pipelined Restore.
- $B_{tape} < B_{net}$: Directly copy from tape to network interface through memory buffer.

5.2. Disk Buffer Management

In the proposed solution, the disk buffer is a resource shared by multiple concurrent archive/retrieve sessions. As discussed earlier, each session may have a different network data transfer rate specified by the QoS requirement of the associated stream. Therefore, assigning the disk buffer I/O resource evenly among them is typically not the most efficient way to maximize network utilization and memory buffer utilization. In addition, enough disk buffer I/O bandwidth should be guaranteed to transfer data from or to the tape drives so the tape drives can work efficiently and reliably in streaming mode. In our proposed disk buffer manager, we treat archive/restore sessions differently based on the network data rates. For simplicity, we describe our scheme with concurrent retrieve sessions from multiple tape drives. A mix of archive and retrieve sessions is a straightforward extension of the all-retrieve case. In addition, due to the burst features of archive and retrieve applications, their mixture is not typical. Our goal here is to provide QoS for different data streams to and from the disk buffer. This requires QoS specification and QoS enforcement mechanisms. Since the data streams of the retrieve sessions alone do not have any QoS specifications, our scheme provides them in terms of an I/O request deadline and release time with the objectives of minimizing tape drive repositioning, i.e., keeping the tape drive in streaming mode, maximizing network bandwidth utilization, and limiting the buffer usage of restore sessions. The first and third objectives require that buffer units in the tape buffer queue be written to disk buffer in time. The second objective requires that the data is read from disk buffer and queued at the network interface in time. Our approach to achieve these objectives is to specify the *release time* and *deadline* for a disk buffer I/O request depending on the hints from the buffer queue length. In addition, our disk buffer I/O scheduler performs scheduling based on the *release time* and *deadline* of I/O requests.

The disk buffer is assumed to be made up of D disks. The buffer is striped across these disks to achieve a high degree of parallelism. If s_{stripe} is the stripe size per disk, then $S_{stripe} = D * s_{stripe}$ is the size of a logical stripe and is used as the basic scheduling unit.

I/O Queuing Model

For each restore session, we maintain two request queues: the Read I/O Queue (RIOQ) and the Write I/O Queue (WIOQ). The queues are made up of requests of size S_{stripe}. In the case of a restore session, the RIOQ contains I/O requests for reading the data from the disk and writing it into the session's network interface buffer. The WIOQ contains requests for reading data from the session's tape drive buffer and writing it onto the disk. At any instant, there is only one read I/O request in the RIOQ, and a new request is added only when the current one is finished. The I/O scheduler works at the granularity of read or write I/Os of size S_{stripe}. These requests get split up into multiple reads/writes of size s_{stripe} and get queued at their corresponding disks. The service order of these disk requests of size s_{stripe} in the disk queue is dictated by the disk scheduling algorithms implemented in disk firmware. Figure 6 shows three concurrent restore sessions r_1, r_2, and r_3. $r_1(a)$, $r_1(b)$, and $r_1(c)$ are three I/O requests of size S_{stripe} for session r_1. These requests get split into smaller requests of size s_{stripe} and get queued at the disk. Here, the disk set is made up of four disks and hence $r_1(a)$ gets split into $r_1(a_1)$, $r_1(a_2)$, $r_1(a_3)$, and $r_1(a_4)$. The scheduling order for requests in a disk I/O queue is not controlled by

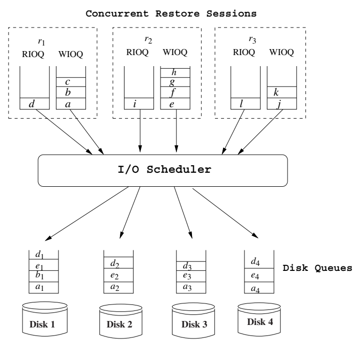


Figure 6. I/O Request Queuing Model

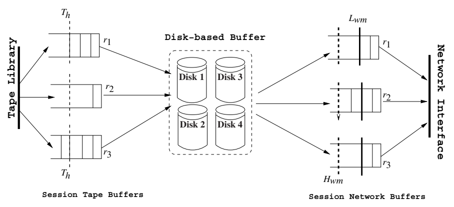


Figure 7. Buffer Queuing Model

our scheduler. The disk is free to schedule requests in any way it wants, as it has more knowledge about the physical geometry and current position of the disk head.

Session-based Network & Tape Buffers

For each restore session, we maintain two in-memory buffers: the Network Buffer and the Tape Buffer, as shown in Figure 7. The network buffer is used to queue data objects that are destined to leave the system through the network interface. The tape buffer is used to queue data objects that are read from tape and are waiting to be written to disk or sent to the network interface. In direct flow mode, data objects read from tape are copied into the tape buffer and later moved to the network buffer directly for transmission across the network. In pipelined mode, data objects read from tape are copied into tape buffer and later moved to the disk for temporary staging before being moved from disk to the network buffer for transmission across the network. Every restore session first starts in direct flow mode and switches to pipelined mode when the size of the tape buffer reaches B_{limit}. For the network buffer, we maintain a low watermark (L_{wm}) and a high watermark (H_{wm}). Both L_{wm} and H_{wm} are used to determine the *release time* and *deadline* for Disk Read I/O requests for this session. For the tape buffer, we maintain a threshold value (T_h) which determines the maximum number of buffer units allowed to be waiting in this buffer queue at any instant. This T_h is used to determine the *deadline* of Disk Write I/O requests for this session.

I/O Scheduler

The job of the I/O scheduler is to decide when to schedule different read and write requests to the disk buffer volume. In the case of a restore operation, the scheduler has to deal with write I/O requests in the WIOQ for writing data that was read from the tape drive and is currently buffered in the tape buffer. In this case, the scheduler also has to handle read I/O requests in the RIOQ for reading data from the disk volume into the network buffer. The problem is that scheduling is difficult because the scheduler has to deal with multiple RIOQs and WIOQs, one for each active session. In our proposed solution, we try to determine the *deadline* and *release time* per request to serve as scheduling criteria.

Write I/O Scheduling

When an object is read from tape into the tape buffer queue of a session, a write I/O request is added to the WIOQ of that session. Let this request be the i^{th} in the queue. The *deadline* for this write I/O request is the latest time when this I/O should be dispatched to the constituent disks so that it can be completed before the $(i+T_h)^{th}$ buffer unit arrives at the tape buffer queue for that session. If the request cannot be completed before the deadline, the tape buffer for the session increases beyond the threshold and overflow occurs. In such cases, the tape drive needs to be told to slow down, which in essence causes the tape to reposition. The period between arrival of the i_{th} request and the $(i+T_h)_{th}$ request with tape bandwidth of T_{tape} is given by,

$$p_{T_h} = \frac{T_h * S_{stripe}}{T_{tape}} \tag{2}$$

Now, the time needed to execute the I/O request of size S_{stripe} by is given by,

$$p_{disk} = t_{dw} + (t_{seek} + t_{rotation}) + \frac{s_{stripe}}{T_{disk}} \tag{3}$$

where t_{dw} is the estimated wait time at the disk due to other pending I/Os and T_{disk} is the disk transfer rate. The *deadline* of the write I/O request is given by,

$$t_{deadline} = t_{current} + p_{T_h} - p_{disk} \tag{4}$$

Read I/O Scheduling

When the network buffer is below the low watermark, data units need to be read from the disk buffer into the network buffer to ensure maximum utilization of the network. The rate at which the network buffer is emptied indicates the available bandwidth of the network interface. Since this rate, or network bandwidth, fluctuates over the lifetime of the session, we use a exponential decay function,

$$T_{net}(i) = \left\{ \begin{array}{l} T_{net}^m(i), i = 0 \\ T_{net}^m(i) * \lambda + T_{net}(i-1) * (1-\lambda), i > 0 \end{array} \right\} \tag{5}$$

Here, λ controls the rate of decay, and T_{net} is the bandwidth of the network interface for the session, or rate at which the network buffer gets emptied. For a session there can be at most one read I/O request scheduled or waiting to be scheduled; a new request is generated only on completion of the previous request. Generation of read I/O requests starts only when a session is switched from *direct-flow* mode to *pipelined* mode. For the purpose of scheduling, if there are K units of data in the network buffer for a session, based on the value of K, one of the following three distinct scenarios could occur. First, when $0 \leq K < L_{wm}$: Network buffer is below the low watermark leading to under utilization of network bandwidth. The solution is to set $t_{deadline} = t_{current}$ for any queued read I/O requests in order to ensure that network buffer is brought back above the low watermark.

Second, when $L_{wm} \leq K < H_{wm}$: Network buffer is between low and high watermarks, i.e., safe state. The solution is to set the deadlines for queued read I/O requests such that it should be completed before network buffer drops below low watermark. The period before network buffer drops below low watermark is given by,

$$p_{l_{wm}} = \frac{(K - L_{wm}) * S_{stripe}}{T_{net}} \tag{6}$$

The deadline for the read I/O request is given by,

$$t_{deadline} = t_{current} + p_{l_{wm}} - p_{disk} \tag{7}$$

T_{net} is recalculated and K is decremented by one every time the buffer unit at the head of the network buffer queue is serviced. If the I/O request is still waiting to be serviced, its deadline is reevaluated with the new K and T_{net}.

Third, when $K \geq H_{wm}$: Network Buffer is above high watermark. The solution is to not schedule this read I/O request. It is marked as un-realizable with a future release time. When the buffer unit at the head of the queue is serviced, K is decremented by one and realizability, and hence the deadline, is reevaluated with the new K value.

Our I/O scheduler uses an Earliest Deadline First scheduling algorithm to dispatch realizable I/O requests belonging to all RIOQs and WIOQs of all active sessions. When any I/O request is selected for scheduling, it is broken down into D smaller requests of size s_{stripe} and dispatched to all disks that form the buffer disk volume. Our scheduler is invoked when an I/O request is generated by any active session or when an I/O is completed by the disks, or when realizability or deadline of a pending I/O request is updated by its owner session.

Once invoked the scheduler tries to schedule all requests that are realizable until the disk queues become full (disks have a fixed size queue, the size of which is fixed by the disk manufacturer to suit the performance characteristics of the disk drive).

6. Autonomous Tape Cartridges

In object storage devices, since we now store metadata along with simple blocks as objects, data has been rendered more autonomous, or self-describing. Now, when objects are being backed up to tape media, we also need to account for backup of metadata or attribute information. We need a way of organizing metadata of objects on tape media so that all high-level software that relies on object metadata information for accessing objects can work efficiently. Moreover, tape cartridges are portable devices. Cartridges can be removed from one object-based ecosystem and inserted into another. In such cases, the tape cartridges should possess enough information about the data that they store such that they can be integrated into the new ecosystem. In this section we describe our approach to making tapes more autonomous. Specifically, we propose an on-tape metadata layout and an indexing/cataloging scheme.

6.1. Metadata layout

When designing any layout on tape media, a couple of key characteristics of tapes play a key role. Rewritable or updatable structures are not supported. Tape is a sequential access medium that does not support update-in-place operations. If one tries to overwrite data at some specific location on tape, any previously written data past the new end of data mark on tape becomes inaccessible. Tapes support search operations using File Marks. A file mark is a universal file separator for tape systems. It is a unique bit pattern created by a specialized write command that is distinguishable from all user data. Tape drives allow the user to locate a file mark without having to read intervening portions of the tape cartridge.

There are three potential strategies to store object metadata. The first option is to have metadata at the *beginning of the tape cartridge*. Even if enough free space is reserved at the beginning of the tape for object metadata, new object metadata cannot be added at a later point in time as tapes do not support in-place update or rewrite of data. The second option is to have the metadata *scattered along with objects*. If metadata is scattered in between data objects, a tape scan to retrieve all metadata takes a long time. This latency can get quite large with high capacity tapes. The last option is to have metadata *following the last object*. Using the search facility provided by the tape drives, efficient metadata access is made possible if all metadata is at the end of last object. However, all metadata needs to be rewritten every time a new object is added to the tape. In order to make this process more efficient, systems generally maintain this metadata in memory and write it back at the end of the tape when it is unmounted. If the system crashes before the write back of this metadata, the tape is rendered useless.

We propose a hybrid tape data layout which is a combination of scattering metadata between data objects and following the last object, as shown in Figure 8. Our layout stores copies of incremental metadata information scattered among data objects to facilitate maximum metadata recovery. The metadata copy at the end of the last object

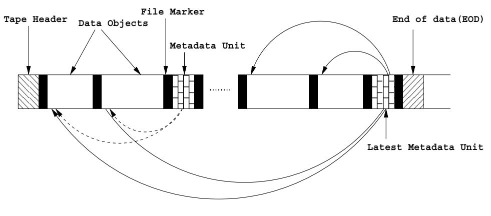


Figure 8. OSD Tape Layout

is the most updated copy with metadata information about all data objects in the tape cartridge. In order to read the latest metadata, one can use the search facility provided by the tape to reach the last file mark which is where the metadata starts. The first unit on the tape is the tape header unit that contains a fixed tape header data structure. This structure includes a magic number to indicate that it is a tape for an Object Archive Tape Device, a version number of the OSD standard that this tape is compliant with, the data block size used by the tape, the partition identifier of the tape, and other device-specific information.Following the tape header unit are object units and metadata units. In our proposed layout, old metadata units are not overwritten (they are left untouched) when appending new objects. This provides a quick way to recover object metadata information in cases where the latest metadata set is lost due to errors.

6.2. Cataloging and Indexing

Due to the large number of objects within a large-scale object storage system and the dynamic migrations of objects between object storage devices, there is a need for an systematical way to keep track of the location of objects. Object Archive Tape Devices further complicate this task since they allow an entire tape cartridge containing many objects to be added into or removed from the system.

6.2.1. Globally Unique Object Identifiers Our first design decision is to keep the object identifier of each object unique within the entire system. This is achieved by letting the MDS (Metadata Server) choose the object identifier for each newly created object. The MDS sends an object creation command along with the chosen object identifier to an object storage device. In addition, since portable tapes can introduce objects created in other systems, we make an assumption that each object storage system will be assigned a unique 16-bit system identifier that will be used as the first 16-bit portion of the 64-bit object identifiers. This leaves every object storage system with 2^{48} object identifiers (more than enough for the foreseeable future).

6.2.2. Translations in Hierarchical Storage In order to access an object given its object identifier, the first step is to locate the object storage device that is storing the object. In object-based cluster file systems, the MDS manages the file system namespace and maps file pathnames of regular files to object identifiers. In the Lustre file system

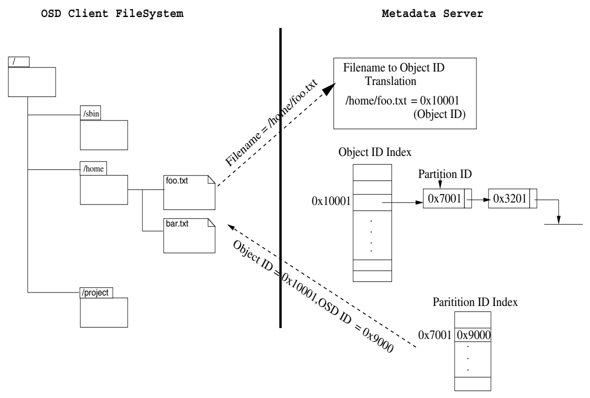


Figure 9. Filename -> (Object ID, OSD ID)

[3] implementation, the MDS also maps file pathnames directly to an object storage device identifier. Nevertheless, this choice makes it inconvenient to migrate or replicate objects to other object storage devices since these operations need to locate and update the file system inode information. In addition, when an object tape containing many objects are unloaded from an object archive tape device or loaded into one, many file system inodes have to be modified. In order to handle these operations efficiently, we propose to manage the object identifier resolution in the MDS, as illustrated in Figure 9. The OSD client file system component queries the MDS to resolve the hierarchical filename to Object ID and OSD ID mapping (a variation of this scheme is also possible with the client itself mapping hierarchical filename to Object ID and querying the MDS to resolve Object ID into OSD ID mappings). We introduce another level of indirection, i.e., partition, to efficiently handle portability of object tapes. As we mentioned previously, each object tape contains one partition with an unique partition identifier. The MDS maintains an index data structure to map every object identifier to a linked list of partition identifiers. Each node in this linked list has a partition number. Using a linked list of partition IDs allows us to have multiple copies of the same object. When an object is requested any one of the online partitions could be used to retrieve the object based on policy decisions.

Furthermore, the MDS maintains another index data structure to map partition identifiers to object storage addresses. With the indirection provided by the partition, the load and unload of object tapes only incurs the update of the mapping from the object tapes' partition identifiers to the object storage device addresses. When a tape is about to be unloaded from an Object Archive Tape Device, it sends the partition identifier of the unloading tape to the MDS. The MDS only needs to update the index of that partition identifier to indicate it is offline. On the other hand, when a tape is loaded into an Object Archive Tape Device, it mounts the tape, verifies the tape is valid, and tells the metadata server the partition identifier extracted from the tape header. If the

Table 1. OSD Configurations

	OSD A Config	OSD B Config
OS	Redhat 9 Kernel 2.4.20	Redhat 9 Kernel 2.4.20
CPU	2 Intel Xeons 2.0GHz	Pentium III 1GHz
Memory	1024 MB DDR DIMM	512 MB SDRAM DIMM
HDD Interface	SCSI Ultra160 SCSI	ATA/133
HDD Spend	10000 RPM	7200 RPM
Avg. Seek Time	4.7ms	8.5ms
Network Interface	Intel Pro/1000MF	3com 3c905c-TX/TX-M 10/100

partition previously belonged to the system, only the index entry of the partition identifier needs to be updated with the new object storage device address. Otherwise, if the tape is totally new to the system, the Object Archive Tape Device reads out the tape metadata from the last unit of the tape and sends the list of object identifiers to the MDS. The MDS creates indices for the object identifiers in addition to the partition identifier.

The hierarchical storage structure is handled by the partition index entries. Each entry has a flag to indicate whether the partition corresponds to online storage or archival storage. When an object is requested, the object identifier is resolved to one or more partitions. If any of these partitions is in online storage, the request will be served from there by returning the object storage address of that online storage device. Otherwise, if all of the partitions belong to archival storage, the MDS selects an online object storage device as the restore target and returns the addresses of the restore target object storage device and the archive object storage device. The client will send an access request for the object along with the address of the archival object storage device to the restore target OSD, which will then initiate an object restore procedure to recall the object.

7. Prototype Evaluation

A variety of performance tests were run to evaluate the performance of OATD under different workloads. In order to demonstrate the effectiveness of the network-aware adaptive scheduling scheme, we have set up different configurations that have the session network bandwidth slower and faster than the tape drive bandwidth. Under each possible combination of the session network bandwidth and the tape drive bandwidth, we measured the data access of the tape drive to verify that it works in streaming mode. We also measured the archive or restore rate or bandwidth observed from the initiating OSD to demonstrate the effectiveness of the pipelining mechanism when it is functioning. In our setup, the OATD node and the OSD A node are connected on the same Cisco Catalyst 4000 Series Gigabit Ethernet switch. The archive/restore session bandwidth between OSD A and OATD is faster than the tape drive bandwidth. The OSD B node is not directly connected on the same gigabit Ethernet switch. Instead, it is connected on a Cisco Catalyst 3500 series XL 10/100 Ethernet switch,

Table 2. Tape Write & Read Bandwidths with Block Size:256KB

	Write Operation	Read Operation
Average Bandwidth (MBps)	12.8054	17.5764
Standard Deviation	0.000548	0.025026

Table 3. Archive and Restore Session Network Bandwidths

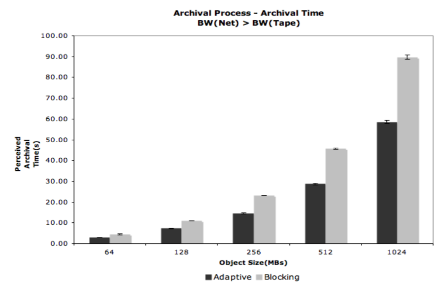
	OSD A	OSD B
Mean Archival Bandwidth (MBps)	18.994	6.809
Standard Deviation of Archival Bandwidth	0.462	0.04
Mean Restore Bandwidth (MBps)	19.777	6.226
Standard Deviation of Restore Bandwidth	0.336	0.432

which is connected to the gigabit Ethernet switch. The session bandwidth between OSD B and the OATD device is slower than the tape drive bandwidth. The configurations of the hosts of OSD A and OSD B are listed in Table 1. The OATD's front-end host connects to a StorageTek TimberWolf 9738 tape library on its SCSI HBA (Host Bus Adaptor). The tape library has one StorageTek T9840A tape drive installed with 10MBps native data transfer rate. We first test the tape drive bandwidth using the standard Unix tool dd. With compression turned on, the tape drive can achieve a higher data transfer rate. The actual data compression rate mainly depends on the compressibility of the data. In order to measure the maximum tape drive data transfer rate on our setup, we use data with all zeros that has high compressibility. From the front- end host of the OATD node, we write/read an object of 2GB with 256KB block size each time. Table 2 shows the average data transfer rate and standard deviation collected from five runs.

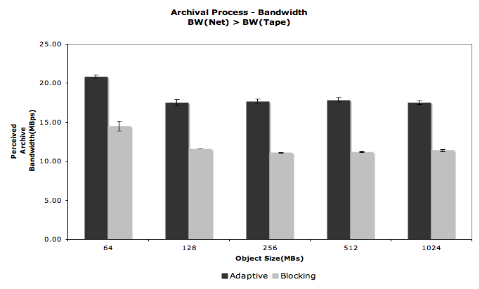
We then measure the maximum achievable archive/restore session data transfer rate from OSD A and OSD B to/from the OATD node, respectively. Since we are interested in how fast data can be sent to or extracted from the memory buffer of the OATD node (maximums), we run an OSD target using a ramdisk device on the OATD node in this testing in order to eliminate tape latencies. A ramdisk device emulates disk drives using main memory. We use a testing tool called 1cmd on OSD A and OSD B to measure object read and write bandwidth. An object of size 8MB is written to and read from the OSD target running on the OATD node. Table 3 summarizes the results. In comparison to the numbers in Table 2, we can see that the tape drive data transfer rate is between the session network data transfer rates of the OSD A and OSD B. All the results described below are averaged over four separate trials and the graphs also indicate the 95% confidence interval about the mean.

7.1. Performance of Pipelined Archive

The archival operation of interest is from OSD A onto the OATD. In our test bench, the network bandwidth between OSD A and the OATD is more than that of the tape drive. In this scenario, the scheduler chooses to use the pipelining archive method. This amounts to staging on the disk and subsequently streaming to tape. In our implementation, the buffer unit size is 5MB, i.e., data is sent to tape for writing when a buffer unit of 5MB has been filled up unless there is no more data in the session. Each session has an upper limit of eight outstanding buffer units in its buffer queue. In this set of tests, as data is received from the network faster than the consumption rate of the tape drive, the buffer queue grows up to eight units and then data starts to be written to the disk buffer. A naive blocking scheme would instead stop accepting network data until more space was available in the buffer queue. Bandwidth estimation is done by monitoring the time required a fill a fixed size buffer. Figure 10 shows the results for archive operation from OSD A onto the OATD. Figure 10(a) shows a comparison of the perceived archive times for the pipelined/adaptive scheme and a naive blocking scheme. Here, perceived archive time refers to the archival time as seen by the archive initiator, which in this case is OSD A. This metric summarizes the latency of the network, the latency of the tape device, and that of the destage operation (only for pipelined scheme). It is clear from the figure that our adaptive approach results in shorter perceived archival times or faster response times. The difference is not that significant in the case of small objects as the buffer used in our implementation itself accounts for 40MB of data. As object size increases, however, the difference in archival time becomes more prominent. For instance, archiving a 1GB object using the blocking scheme takes about 90 seconds where using our adaptive approach takes about 59 seconds. This amounts to a reduction of at least 35% in the archival time for objects of size greater than a gigabyte. Figure 10(b) shows the comparison of perceived archival bandwidth for the same two schemes. Clearly, our adaptive approach helps achieve a higher archival bandwidth than the blocking scheme. Perceived bandwidth for larger objects (128-1024MB) is almost constant. This constant value turns out to be the disk buffer saturation point, and hence an indirect reflection of the maximum tape drive bandwidth. This clearly shows that the tape drive is working in streaming mode. However, for smaller objects (8-64MB), the perceived bandwidth appears to be higher as the disk buffer capacity limits are not reached, and hence all the data is getting buffered in disk first before it is dispatched to the tape, which amounts to naive staging. Also, the tape drive bandwidth for both the schemes was found to be constant (about 17MBps) and hence not shown here. This is expected because in both the schemes, data is always ready to be written to tape, either from the disk (pipelined) or from the network itself (blocking).



(a) Comparison of Archival Time



(b) Comparison of Perceived Bandwidth

Figure 10. Archival from OSD A to OATD

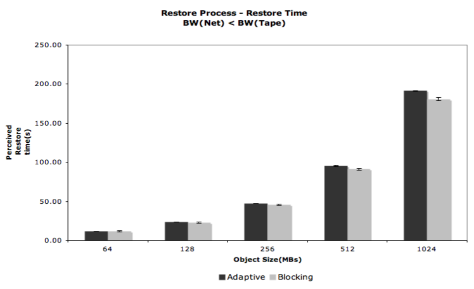
7.2. Performance of Pipelined Restore

The restore operation of interest is the scenario of restoring an object from the OATD onto OSD B. In our test bench, the network bandwidth between OSD B and the OATD is less than the tape drive bandwidth. Hence, the scheduler here chooses to use the pipelined restore scheme. Figure 11 shows a comparison between the pipelined/adaptive scheme and the blocking scheme. In the blocking restore method, when there are eight outstanding buffer units in the restore session's buffer queue waiting to be transferred, the tape drive stops and waits for an available slot in the buffer queue. Figure 11(a) shows the comparison of perceived restore times for both the blocking scheme and the adaptive scheme. These results are a little counter intuitive. One would expect the restore time to be a little better for the adaptive scheme when compared to blocking scheme or equal at worst. Two important factors influence this result. One is the repositioning time of the tape drive which adds more delay to the restore time in the case of the blocking scheme. The other factor is that the operation of copying data first into the disk from the tape buffers and then copying it back from disk into the network interface buffers adds more delay to the restore time in the case of the adaptive approach. The latter factor seems to outweigh the former; hence, the restore time for adaptive approach seems to be slightly higher (about 2%) than that for the blocking scheme. However, the restore time is not the only focus of the pipelined restore scheme. The main focus is to use the tape drive in streaming mode so that the number of repositions is smaller (leading to less degradation of tape) and the active window for any particular restore operation is as small as possible such that the tape drive is free to service other requests (mutual exclusion). For this purpose, we measure the bandwidth of the tape drive for any particular restore operation. This is an indirect measure of repositions. The higher the tape bandwidth, the lesser the number of repositions, and vice versa. Figure 11(b) shows the comparison of tape drive bandwidths for both adaptive and blocking schemes. It clearly shows that our adaptive scheme keeps the tape drive streaming at a constant rate. For small objects (8-64MBs) the difference between the two schemes in terms of tape bandwidth is not significant as in most of these cases the buffers we use in our implementation (40MBs) absorb all the requests. But for large object sizes the difference is significant. For instance, for a object of size 1GB, the adaptive scheme helps maintain a constant tape drive bandwidth of about 14.2 MBps compared to about 6MBps with the blocking scheme. In terms of the tape drive time used by this restore operation, in the case of the adaptive scheme, the tape drive is kept busy for about $1024/14 = 73$ seconds whereas the blocking scheme kept the tape drive busy for about $1024/6 = 170$ seconds. Hence, we can conclude that the adaptive scheme helps achieve the same restores in about 55-60% less tape drive time when compared to blocking schemes for objects of size greater than a gigabyte. Also, the network bandwidth of the perceived restore bandwidth seems to be constant across methods and hence it not reported here. This is to be expected as the bottleneck is still the network and latency of the whole operation is determined by the latency of the slowest component. For more detailed experimental valuation results and simulation evaluation can be found at [15].

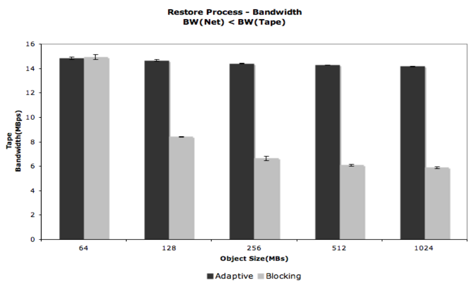
8. Conclusion

In this work, we investigated the features and design of OSD based on tape devices. This study provides the essential components for constructing large-scale storage hierarchies. While there are possibilities to integrate tape devices using the existing block interface or file interface, we believe that an object interface gives tape devices the most smooth and efficient integration with the rest of the system. More specifically, the advantages include a consistent security mechanism, uniform object representation, and abstracted tape details. In this study, we propose extensions to the object storage interface that address the needs of object archiving applications as well as the sequential-access characteristics of tape devices. The object archive/retrieve operations cover not only the object data but also object attributes that are an indispensable part of objects.

Our proposed intelligent buffering scheme consistently outperforms traditional techniques like disk staging that have been used before to overcome bandwidth mismatches. The performance evaluation of the prototype confirms the



(a) Comparison of Restore Time



(b) Comparison of Tape Bandwidth

Figure 11. Restore from OATD onto OSD B

same. Our proposed approaches to making tape cartridges more autonomic are very efficient and easy to implement. We believe that such mechanisms will be the key to building next generation storage systems. In conclusion, we hope that this work serves as a good initial point for tape system designers of next generation storage solutions.

9. Acknowledgments

We would like to thank all our sponsors - DISC member companies, Department of Energy's Los Alamos National Labs and National Science foundation for their continued support.

References

[1] G. A. Gibson, D. F. N. amd Khalil Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka, "A cost effective, high bandwidth storage architecture," in *Proceedings of ACM SIGPLAN Notices*, 1998, pp. 92–103.

[2] M. Mesnier, G. Ganger, and E. Riedel, "Object based storage," *IEEE Communications Magazine*, vol. 41(80), pp. 84–90, 2003.

[3] *Lustre: A Scalable High Performance Cluster File System*, Cluster File systems.

[4] R. O. Weber, "Scsi object based storage device commands - 2(osd-2)," T10 Working Draft, Tech. Rep., October 2004. [Online]. Available: http://www.t10.org/ftp/t10/drafts/osd2/osd2e00.pdf

[5] "Data storage devices and systems roadmap," Information Storage Industry Consortium, Tech. Rep., January 2005.

[6] A. L. Drapeau and R. H. Katz, "Striping in large tape libraries," in *Proceedings of Supercomputing*. Portland, Oregon: IEEE Computer Society Press, 1993, pp. 378–387.

[7] A. Drapeau and R. Katz, "Striped tape arrays," in *Proceedings of Twelth IEEE symposium on Mass Storage Systems*, Monterey, CA, April 1993, pp. 257–265.

[8] L. Golubchik, R. R. Muntz, and R. W. Watson, "Analysis of striping techniques in robotic storage libraries," in *Proceedings of 14th IEEE Symposium on Mass Storage Systems*. IEEE Computer Society Press, 1995, pp. 225–238.

[9] S. Christodoulakis, P. Truantafillou, and F. Zioga, "Principles of optimally placing data in tertiary storage libraries," *VLDB*, pp. 236–245, 1997.

[10] J. Li and S. Prabhakar, "Data placement for tertiary storage," in *Proceedings of 19th IEEE/10th NASA Goddard Conference on Mass Storage Systems and Technologies*, April 2002, pp. 193–208.

[11] B. K. Hillyer and A. Silberschatz, "Random i/o scheduling in online tertiary storage systems," in *Proceedings of ACM SIGMOD international conference on Management of data*, Newyork, USA, 1996, pp. 195–204.

[12] S. Prabhakar, D. Agarwal, and A. E. Abbadi, "Optimal scheduling algorithms for tertiary storage," in *Distributed Parallel Databases*.

[13] M. Lijding, P. G. Jansen, and S. J. Mullender, "An efficeint real time tertiary storage schedular," in *Proceedings of 21st IEEE/12th NASA Goddard Conference on Mass Stoarge Systems and Technologies*, Maryland, April 2004, pp. 245–260.

[14] X. Zhang, D. Du, J. Hughes, and R. Kavuri, "Hptfs: A high performance tape file system," in *Proceedings of 14th NASA Goddard/23rd IEEE conference on Mass stoarge System and technologies*, College Park, MD, May 2006, pp. 275–288.

[15] D. He, N. Mandagere, and D. Du, "Implementation of network aware object based tape device," Digital Technology Center, Univ of Minnesota, Tech. Rep., 2007. [Online]. Available: http://www.dtc.umn.edu/disc/publications.shtml