NASA

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.

4. Title and Subtitle
Performance Analysis of the Unitree Central File

5. Report Date

6. Performing Organization Code

7. Author(s)
Odysseas I. Pentakalos

8. Performing Organization Report No.

10. Work Unit No.

9. Performing Organization Name and Address
University of Maryland Baltimore County
302 Administrative Building, 1000 Hilltop Circle
Baltimore, Maryland 21250-5394

11. Contract or Grant No. NAS5-32337
USRA subcontract No. 5555-24

12. Sponsoring Agency Name and Address
National Aeronautics and Space Administration
Washington, DC 20546-0001
NASA Goddard Space Flight Center
Greenbelt, MD 20771

13. Type of Report and Period Covered Final
August 1993 - May 1994

14. Sponsoring Agency Code

15. Supplementary Notes
This work was performed under a subcontract issued by
Universities Space Research Association
10227 Wincopin Circle, Suite 212
Columbia, MD 21044 Task 24

16. Abstract

This report consists of two parts. The first part briefly comments on the documentation status of two major systems at NASA's Center for Computational Sciences, specifically the Cray C98 and the Convex C3830. The second part describes the work done on improving the performance of file transfers between the Unitree Mass Storage System running on the Convex file server and the users workstations distributed over a large georgraphic area.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
hierarchical mass storage systems	Unclassified--Unlimited

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	1	

NASA Form 1626 Oct 86

Final Report to NASA on ALIBI

David Flater, CESDIS

4/22/94

SUMMARY

ALIBI (Adaptive Location of Internetworked Bases of Information) succeeds in locating and retrieving information over the Internet without the use of centralized resource catalogs, navigation, or costly searching. Its powerful query-based interface eliminates the need for the user to connect to one network site after another to find information or to wrestle with overloaded centralized catalogs and archives. This functionality was made possible by an assortment of significant new algorithms and techniques, including classification-based query routing, fully distributed cooperative caching, and a query language that combines the practicality of Boolean logic with the expressive power of text retrieval. The resulting information system is capable of providing fully automatic resource discovery and retrieval access to a limitless variety of information bases.

1 Introduction

The Internet has been one of the most important contributions to the modern computing environment. It has had a considerable impact on the way that many individuals and companies conduct their daily business. Although it has remained fairly stable despite an exponential rate of growth, some applications have suffered more than others. One application that has suffered quite a lot is the use of the Internet as a source of information – what some would call a digital library.

When the number of public archives on the net became large, the resource discovery problem arose[1]. Users found themselves navigating file system after file system with FTP in search of specific files and having diminishing success at finding them. Riding to the rescue was Archie[2, 3], a centralized catalog that simply compiled the names of files in the directory trees of participating archives. Users who knew the name of the file they wanted could search Archie to find an accessible copy, then use FTP to retrieve it.

As the Internet continued to grow, Archie began to bog down. The volume of data that needed to be catalogued and the volume of queries over that data made rapid

response impossible during peak usage, even with replication of the Archie database. However, the more significant problem was with the database itself. It contained only filenames. Although Archie helped users to find files once they knew the names of those files, it did not support any kind of content-based searching. With the huge volume of data available on the Internet constantly growing, it became more and more likely that any given piece of information was out there *somewhere,* but we lacked the ability to find it.

A complete solution to this problem has not yet appeared in the Archie system. A *whatis* database was added that permitted centralized indexing of software with brief descriptions, but it does not seem to have reached critical mass. At any rate, given the tendency of the filename database to bog down, we can easily imagine how much worse the situation would be if every file in the filename database also had an entry in the *whatis* database.

Centralized indexing cannot be the answer to the resource discovery problem in the Internet. Although its growth will eventually slow from an exponential to a geometrical rate once the number of Internet addresses per capita reaches the limit of convenience, it is already too big for centralized indexing and it is still growing exponentially. Decentralized tools like WWW[4, 5], Gopher[6] and WAIS[3, 7] resulted in part from our experiences with Archie. Unfortunately, in trying to solve some of the fundamental problems faced by decentralized systems, many of the new tools have re-introduced centralized indexing in some form or another. Each of them has been successful so far, but the indices are growing, and the costs are again mounting.

With this work, we offer a new alternative to centralized cataloging of resources, navigational resource discovery, direct manual file transfer, and massive centralized archives. Our alternative is named ALIBI, for Adaptive Location of Internetworked Bases of Information. To support networked resource discovery and information retrieval without resorting to any of the problematic techniques that have limited other systems, it was necessary for us to develop an entirely new approach. A combination of novel algorithms and careful design, developed in advance and improved along the way to the final implementation, made possible the software system we now have.

ALIBI consists of a network of information servers and a collection of information bases. The information bases are affiliated with individual information servers. The network of ALIBI, running as an application layer above the Internet, is called the Übernet. The servers are called Unetds (Übernet daemons). ALIBI offers all of the following services in a single system:

- Completely Automatic Resource Discovery

  The information servers accept queries from users (via a client program) and either arrange for them to be answered by the information bases affiliated with them or forward the queries to other servers that might be able to do so. Users

provide queries that describe what they want, and the information system attempts to locate and fetch it. Users do not need to indicate the source of the information they desire, traverse a hierarchical file system, or navigate a hyperdocument to get to the information. Large, centralized archives and meta-indices are unnecessary because ALIBI locates the data wherever they lay without user assistance.

- Fully Distributed Resource Discovery

  ALIBI finds information without the use of any centralized services or algorithms. A special, fully distributed point-to-data routing algorithm is used by each site to forward queries towards the data they seek[8]. Neither broadcasting of queries nor broadcasting of metadata are required to support the routing of queries. The information used to route queries is gleaned from resource providers and from response messages as they are passing through on their way to the sites that requested the information. Direct contact with distant sites is not needed.

- Cooperative Caching

  Some of the waste and redundancy present in file transfers today is eliminated by ALIBI's resource discovery mechanism, since it is likely to find the closest available source of data. Much of the remaining waste and redundancy is eliminated by ALIBI's integrated cooperative caching[9]. Not only are retrieved data cached for reuse at the same site, they are cached cooperatively by groups of sites for the common good of all. The fully distributed caching mechanism uses cache space at underutilized sites to help out sites that are overburdened. The administrator at each site is free to choose how much cache space to provide; the system automatically adjusts to tolerate imbalances.

- Arbitrary Topology Information Network

  The list of sites hosting the information servers with which the local server should attempt to communicate must be provided by the local administrator. However, providing this list should not cause stress to anyone. It is not necessary to update the lists on both sides of a connection for the connection to be established; neither is it necessary to update the list whenever neighboring sites (those with which there is direct communication) appear or disappear. All this is sorted out automatically and transparently by the servers. Both the point-to-point and point-to-data routers are topology-independent. Although it is recommended that geographically nearby sites are chosen as neighbors in the information network, the administrator has complete freedom to choose

whichever and however many sites he or she sees fit. Connections that offer poor performance are simply disused by the information system.

- Self-Maintaining Information Network

  The query router and other ALIBI components automatically adapt to shifting network topology. Broken connections are taken in stride, and periodically the effort is made to re-establish them. The abrupt appearance and disappearance of information servers does little to disrupt the normal operation of the information system as a whole. Unlike WWW/Mosaic and Gopher, ALIBI is not phased by a missing information base. It simply locates an alternate without bothering the user.

- Optimized Information Flow

  ALIBI routes response messages along the paths of least delay. The routing of query messages is patterned after the routing of response messages to the extent that the same low-cost paths are used. Since these paths conform to the characteristics of the underlying network and not to the unrelated topology of the information space, queries do not zigzag across long distances to locate data. Because of the flexibility of ALIBI's routing mechanisms, ALIBI is sometimes able to move data more reliably than direct FTP or telnet by avoiding unreliable chunks of the network.

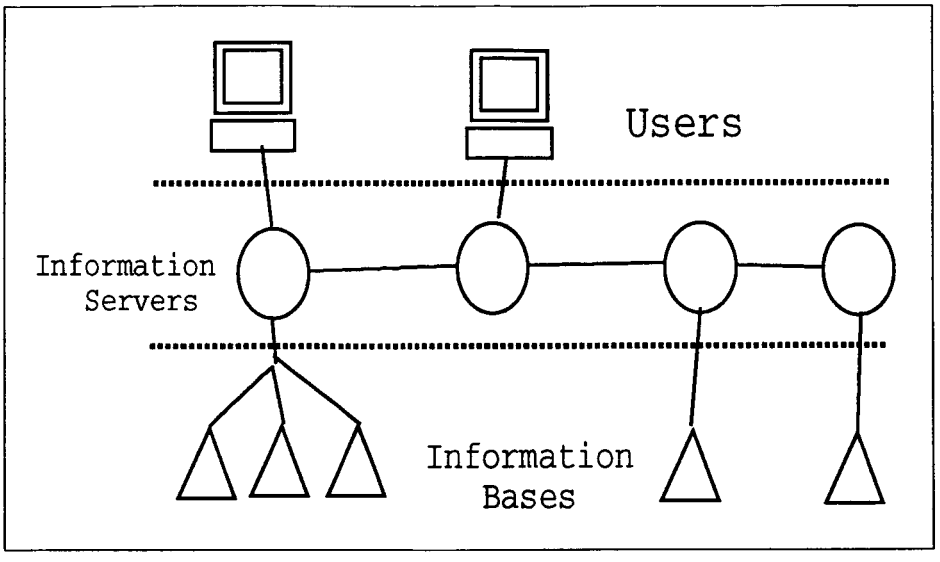
2 The ALIBI Information System

2.1 Resource Model

A key step in the development of ALIBI was determining a resource model. We needed to decide exactly what was the best way to partition the work between the general-purpose information servers and the domain-specific mediators[10] that would access individual information bases. Our goal was to find an orthogonal, flexible model that would simplify the task of installing new mediators as more diverse information bases became available, but without placing unnecessary restrictions on the nature or complexity of the mediators.

The format of queries given to ALIBI was originally a list of keywords or simple English. A "junk words" list was used to eliminate irrelevant English verbiage from the query, a global thesaurus was used to replace query terms with standardized equivalents, and the query was then passed to the mediators to match as many keywords as possible using whatever mechanisms they chose to employ. As the system grew larger and we began to think about how to incorporate some non-fuzzy Boolean

Figure 1: ALIBI's Layered Information System Model



logic into the query language, it became clear that our approach was inadequate. Too much work was being done by the servers. Thesaural translation is a task that must be done within the specific domain of a particular information base, not at the global level. Furthermore, it is unsafe to remove "junk words" from a query in a global information system such as ALIBI. They might be acronyms for something important, or they could be significant in another language.

Since it was obvious that major changes would be needed, we scrapped our existing interface and rebuilt it based on a coherent resource model. This investment of time and effort quickly paid for itself with much easier development and installation of mediators.

Our resource model requires the following definitions:

1. Mediators

   A mediator is an entity that communicates with an information server to provide access to a single logical resource (information base). The mediator might actually coalesce several physical resources into a single logical resource, but this action is transparent to the information server. Each information server may be connected to any number of mediators, but there would be little advantage in having a mediator communicate with multiple information servers when those servers can communicate with one another.

2. Object Identifiers

   An object identifier, or OID, uniquely identifies a single datum in the entire information system. The mediator for the information base containing that datum must be able to retrieve the datum when presented with its OID. For the purposes of the resource interface, the OID would only need to be unique within each resource, since it is not necessary for any external components to be able to locate the resource given only an OID. However, OIDs have other uses that do require global uniqueness.

3. Subqueries

   A subquery is a list of keywords that can be processed by the mediator to yield the set of OIDs identifying data in its information base that "match" the subquery.

4. Matching

   A datum matches a single keyword if it has a very high degree of relevance to that keyword according to the domain-specific logic used by the mediator. A datum matches a subquery if it has a very high degree of relevance to *each* of the keywords in the subquery. This insures that adding more terms to a subquery will make it more specific.

5. Queries

   A query is something that is processed by an information server, not by a mediator. Mediators only need to respond to subqueries. The query language understood by the information server can be arbitrarily complex.

In theory, one could simplify matters even more by defining subqueries to be single keywords and forcing the conjunctive logic to be performed by the information server. In practice, this would be extremely inefficient for large information bases, and any semantics that the information base might give to associations between several keywords could no longer be useful.

A resource is modeled as an entity that provides the following minimal set of services:

1. Process a subquery

   When presented with a subquery, the mediator must return a set of OIDs that identify matching data.

2. Fetch and classify a datum

   When presented with an OID that it earlier returned in response to a subquery, the mediator must return the datum that the OID identifies along with an accurate classification of that datum.

3. Declaration of class (optional)

   The resource may inform the information server of the class of data it contains. The information server can then save time by not processing queries against resources that contain the wrong kind of data.

This minimized resource model is powerful enough to support many non-trivial forms of information retrieval, including that of ALIBI. The query language processed by ALIBI information servers is described in the next section.

2.2 Processing and Routing of Queries

Our query language has the rare quality of combining fuzzy and non-fuzzy semantics in a coherent manner. It was designed to allow simple implementation, rather than user-friendliness, but it would not be a major task to have the client translate from a more intuitive format. The elements of the language are:

- Subqueries between parenthesis: `(cache software)`
- Individual OIDs in brackets: `[object-key host-id object-version]`
- Binary AND operator: `&`
- Binary OR operator: `|`
- Unary NOT operator: `~`

Expressions are built with postfix ordering[11], so disambiguating parenthesis are not needed. The ability to specify particular OIDs in queries was included specifically to support the implementation of a **more** command in the client. If the datum identified by `[OID]` were returned in response to `(sound)(index)~&` and the user typed "more," the query `(sound)(index)~&[OID]~&` would automatically be submitted. Repeated use of the **more** command adds additional clauses to the query until finally there are no more data to be returned and the query fails. This automatic query construction gives the user a simple way to retrieve more data like the one just retrieved while preserving the simplicity of the query processor.

To process a full query, the information server sends the subqueries to the information base to be replaced with sets of OIDs, performs Boolean operations on these

sets to produce a single set of OIDs, and, if the resulting set is not empty, instructs the information base to retrieve one of the data identified by an OID in the set. If there are multiple OIDs in the final set, one is chosen at random by the information server. The user is free to retrieve the others using the **more** command.

Accurate routing of queries is accomplished by sending queries towards sites that have answered similar queries in the past. The query classification algorithm, which determines this similarity, operates by examining the query for keywords that it recognizes. Data classes in ALIBI are specified by lists of keywords having decreasing significance, such as "blob software msdos." For this example, "blob" is the major class of data.[1] "Software" is a type of blob, and "msdos" is a type of software. Each word thus adds additional specificity to the more general class preceding it. Classes can be generalized by simply removing words from the end. Although a hierarchical class structure has its limitations, we were forced to make a decision when ALIBI was ready for implementation, and this is what we chose as the most practical approach.

"U" is defined to be the universal class, of which all others are a subclass. All classes except "U" can be considered to have an implied "U" at the beginning, and sufficient generalization of any class eventually results in "U."

2.3 Mediation

Each information base or resource is made available to ALIBI by a mediator connected to a single Unetd. Whatever operations are necessary to translate ALIBI subqueries into queries that are understood by the information base are done by the mediator. The protocol spoken by Unetd to its mediators is extremely simple, consisting of a RESET command, a FETCH command, and subqueries that are just lists of keywords. Mediators respond to subqueries and FETCHes with lists of object identifiers and raw data. They also have the ability to declare the specialty (class) of the provided resource to Unetd so that they will not be bothered with irrelevant queries.

In order to open a communication channel with the information server, the mediator creates and opens two named pipes in the directory used by Unetd.[2] Unetd periodically scans its directory for named pipes. When it finds them, it opens them and immediately unlinks them. The pipes then disappear from the file system, but continue to exist as channels of communication through memory. The resource is then available to Unetd.

OIDs are unique in the entire information system because they are constructed with three fields: an object key provided by the information base, the network address

[1] Blob is an acronym for "Binary Large OBject." The creator of this acronym is unknown to us, but it is used by some SQL databases that have multimedia support.

[2] The semantics of Unix pipes force us to open both of them in "read-write" mode to avoid being blocked, even though one is used exclusively for reading and the other for writing.

of the site running the information server connected to the information base, and a version number or timestamp used to distinguish older and newer versions of the same data object. An example of an OID is

```
[/home/faculty/alibi/blobs/sound/00-index.txt -2108333000 754087762]
```

The first field, the object key, in this case is the full path name of the file being referenced. The resource that produced this OID is a generic "blobs" resource that simply treats entire files as individual data, so the full path name of the file makes a perfect object key. The next field is the Internet address of the machine running the server, daisy.cs.umbc.edu, output as an integer. The last field, the version number or timestamp, is the time at which the file was last modified according to the Unix **stat** function, expressed in seconds since 00:00:00 GMT 1/1/70 as is the Unix tradition. The format of the first and third fields will vary from one information base to the next, but the interpretation is the same: the first field is character data that identifies the data object, the last field is a number such that a higher number in the third field with the first and second fields being identical indicates a newer version of the same datum.

3 Example Resources

3.1 Cache Resource

The cache resource is always at the head of the resource list and the first to be checked for needed data. It is not an external process, but in most other respects it appears to the resource manager like any other resource. From the perspective of the resource manager, the only special thing about the cache is that a different function call is used to communicate with it. Although the fact that the cache has type "U" instead of something more specific is unusual for a resource, it is not inconsistent.

The function call provides one parameter to the cache resource that is not given to remote resources: the classification of the query. Most resources do not need this since they declare themselves to have a specific type and they are guaranteed not to be bothered with queries that are not even close. The cache, on the other hand, has no particular type but needs to match the query with replicas (cached responses) that are of the correct class. Most resources have the advantage of domain-specific knowledge to assist with matching queries to data; the cache has no such advantage.

When presented with a subquery, the cache returns the OIDs of all replicas with a matching class whose descriptions contain every keyword in the subquery, with the exception that keywords constituting part of the query's class are considered to have been matched already. It would be risky to use a fuzzy matching algorithm

Figure 2: A Session with ALIBI

```
Enter a query now, or enter 'quit' to quit.
(msdos cache software)

Waiting for reply....
Response created at zing.ncsl.nist.gov Sun Apr 17 09:25:24 1994
Response path: zing.ncsl.nist.gov topdog.cs.umbc.edu greyhound.cs.umbc.edu
This datum has been in cache since Fri Apr  8 13:09:08 1994
Response created at retriever.cs.umbc.edu Fri Apr  8 13:08:40 1994
The following datum was provided by a blobs resource.
ID: /net/wuarchive.wustl.edu/archive/systems/ibmpc/msdos/diskutil/pckwk319.zip
 -2108333024 614736000
Class: blob software msdos diskutil
Description: pckwk319 zip  B   16358  890625  PC KWIK shareware disk cache
 program  v3 19 diskutil
Binary data follows ... 16358 bytes, filename pckwk319.zip

Warning:  existing files will be appended without further warning
Choose filename in which to save response [pckwk319.zip]:
Saving response....
Query path: greyhound.cs.umbc.edu zing.ncsl.nist.gov
Start time: 766589093
Answer for 185 at greyhound.cs.umbc.edu
Have a nice day =|-)

Enter a query now, or enter 'quit' to quit.
more
(msdos cache software)[/net/wuarchive.wustl.edu/archive/systems/ibmpc
  /msdos/diskutil/pckwk319.zip -2108333024 614736000]~&

Waiting for reply....
Response created at retriever.cs.umbc.edu Sun Apr 17 09:25:37 1994
Response path: retriever.cs.umbc.edu greyhound.cs.umbc.edu
The following datum was provided by a blobs resource.
ID: /net/wuarchive.wustl.edu/archive/systems/ibmpc/msdos/diskutil/cacheart.zip
 -2108333024 588816000
Class: blob software msdos diskutil
Description: cacheart zip  B   13858  880829  Review of disk cache programs
 diskutil
Binary data follows ... 13858 bytes, filename cacheart.zip

Warning:  existing files will be appended without further warning
Choose filename in which to save response [cacheart.zip]:
Saving response....
Query path: greyhound.cs.umbc.edu topdog.cs.umbc.edu zing.ncsl.nist.gov
 sunset.ncsl.nist.gov retriever.cs.umbc.edu
Start time: 766589126
Answer for 185 at greyhound.cs.umbc.edu
Have a nice day =|-)

Enter a query now, or enter 'quit' to quit.
quit
```

for the cache since it is general-purpose; it is not risky for other resources to use domain-specific thesauri to help with matching.

The class of a query is considered to match the class of a replica if the query class is the same as or a generalization of the class of the replica, or if either class is "U."

3.2 Blobs Resource

For our first real resource, we implemented a generic blobs resource that could be used to turn any collection of indexed files into an information base. We have used this mediator with an NFS connection to provide the MS-DOS subtree of wuarchive.wustl.edu, using the index files that were already there. The only customizations were to parse the slightly different format of wuarchive's index files and to generate resets when repeated NFS failures occurred. Isolated retrieval failures are assumed to mean that the index is out-of-date, and the relevant index entries are deleted. Repeated failures indicate an NFS outage, so the mediator informs Unetd that it is "down" until it succeeds in establishing a new connection and rebuilding its index.

We are also using the blobs resource to provide a group of image files at NASA GSFC. This time, we needed to customize the mediator to add some additional lines to response messages that explain the copyrights on the files that were generated by employees of IBM. A collection of sound files at UMBC and an archive at NIST are being provided using the unmodified mediator.

Since the blobs resource is generic, non-fuzzy retrieval is the default. To generate internal descriptors against which to match query terms, the complete path and name of the file and any descriptions found in indices are concatenated and all punctuation marks are removed. When the blobs resource is applied within a domain, as we have applied it to MS-DOS software at wuarchive, the stock retrieval mechanism can be upgraded to a domain-specific fuzzy retrieval method. MS-DOS and NASA thesauri are on the agenda when time becomes available.

The type that a blobs resource declares for itself ("blob software msdos" for wuarchive) is augmented by the directory names in the path to a file in order to form the classification for that file. The path is relative to whatever directory is specified as the "root blob directory." For wuarchive via NFS, /net /wuarchive.wustl.edu /archive /systems /ibmpc /msdos is the root blob directory. A file appearing in /net /wuarchive.wustl.edu /archive /systems /ibmpc /msdos /gnuish is therefore classified as "blob software msdos gnuish."

Figure 3: Header comments from ppmtogif.c

```
/* ppmtogif.c - read a portable pixmap and produce a GIF file
**
** Based on GIFENCOD by David Rowley <mgardi@watdscu.waterloo.edu>.A
** Lempel-Zim compression based on "compress".
**
** Copyright (C) 1989 by Jef Poskanzer.
**
** Permission to use, copy, modify, and distribute this software and its
** documentation for any purpose and without fee is hereby granted, provided
** that the above copyright notice appear in all copies and that both that
** copyright notice and this permission notice appear in supporting
** documentation.  This software is provided "as is" without express or
** implied warranty.
**
** The Graphics Interchange Format(c) is the Copyright property of
** CompuServe Incorporated.  GIF(sm) is a Service Mark property of
** CompuServe Incorporated.
*/
```

3.3 Usenet News Resource

The Usenet news resource turns newsgroups into information bases. Since we did not have direct access to the news spooling directory at any site, we implemented this resource using NNTP (Network News Transfer Protocol). All information is fetched remotely from an NNTP server. As ALIBI grows, we hope that we will be allowed to access one or more news spools directly to enhance performance.

Query keywords are matched against descriptions that contain the subject lines from news articles along with the names of the newsgroups to which those articles belong. Matches between subqueries and descriptions are made by the same algorithm used in the blobs resource. General queries match the description words derived from the names of newsgroups; specific queries match the description words derived from subject lines. The internal index of descriptions is periodically refreshed to keep up with the appearance of new articles and the expiration of old ones.

OIDs are formed by making the first field a combination of newsgroup and article number and making the version always zero. The version is not necessary because netnews assigns article numbers in a serial fashion, even when a new article supersedes an old one. Our decision to combine article number and newsgroup to form the ID instead of using the unique article IDs assigned by netnews was a matter of

programming convenience and efficiency. It would not make much sense in ALIBI to have multiple sites providing the same newsgroups, so there is no harm in using this kind of OID.

Classifications for articles are formed by prepending "news" to the list of words in the name of the applicable newsgroup. For example, all articles in alt.politics.correct are classified as "news alt politics correct." We adopt the netnews hierarchy as a subtree of our classification system in the same way we adopted the directory structure of blob archives.

3.4 Software Reuse Library

As an example of how diverse kinds of retrieval can be supported by ALIBI, we modified the blobs resource to create a primitive software reuse library for C language source code. In the past, work has been done to support the retrieval of source code using variations on methods employed by full-text retrieval systems[12]. To avoid the need to re-implement these complex algorithms for the sake of a single, prototype mediator, we settled for the simple heuristic of extracting the first nontrivial line of a `*.c` or `*.h` file to serve as a descriptor. Quite often, the comments at the head of a program begin with a one-line description of the program. For example, the beginning of the `ppmtogif.c` source file from the PBMPLUS distribution is shown in Figure 3.

The line "ppmtogif.c - read a portable pixmap and produce a GIF file" is read by the mediator and used to index `ppmtogif.c`. Although this heuristic only works when the programmer follows the convention of starting the program with a one-line description, it suffices for the purpose of demonstrating ALIBI's ability to support software reuse. It also serves as a trivial example of how automatic indexing can be used by ALIBI resources having specialized domains.

The classification of data and assignment of OIDs by the source code resource are identical to what is done by the blobs resource.

3.5 SEC EDGAR Resource

EDGAR is a public FTP archive that was recently opened at town.hall.org. It contains the electronic versions of forms filed with the SEC (Securities and Exchange Commission) by companies in the U.S. The names of the files containing the forms are cryptic, but the files are indexed by company name in a separate index file. We modified the blobs resource to access the EDGAR database with anonymous FTP.

When it is reset, the Edgar mediator FTPs the index and reads it into memory. Subqueries are then matched against company names. Matching files are retrieved

Figure 4: Sample Response from FIPS 55-2 Geographical Database

```
Response created at sunset.ncsl.nist.gov Tue Mar  8 13:53:19 1994
Response path: sunset.ncsl.nist.gov cesdis7.gsfc.nasa.gov
 dunloggin.gsfc.nasa.gov greyhound.cs.umbc.edu retriever.cs.umbc.edu
The following datum was provided by the NIST FIPS 55-2 Geographical
 Database (Virginia)
Thanks go to Henry Tom for this database.
ID: 6911 -2130298111 0
Class: government USA census geography VA
Description: Wolf Trap Farm Park For The Performing Arts
FIPS 55-2 Record:  5187256VA 1 1M4Wolf Trap Farm Park For The
   Performing Arts  059Fairfax          22180           8840 1110
A brief summary of the significant features of this record follows.
-     The place is called Wolf Trap Farm Park For The Performing Arts
-     It's a federal facility
-     It is in the state of VA
-     Its county is Fairfax
Query path: retriever.cs.umbc.edu cesdis7.gsfc.nasa.gov
 sunset.ncsl.nist.gov
Start time: 763152827
Answer for 16 at retriever.cs.umbc.edu
Have a nice day =|-)
```

using FTP and given to the Unetd as blobs. Although the information is all textual, most of the forms are large enough that it is better to treat them as blobs.

All data fetched from the EDGAR database are classified as "government USA SEC EDGAR." OIDs consist of the relative pathname of the file, the host ID, and a null version number since individual forms are never altered.

3.6 Geographical Database

Henry Tom of NIST provided us with a geographical database conforming to FIPS 55-2. The format specified by FIPS 55-2 is a flat file containing 132-character records divided into fixed-length fields. The Edgar mediator was the basis for the FIPS 55-2 mediator since it had already been substantially simplified from the blobs mediator.

The FIPS 55-2 mediator matches subqueries against place names and returns individual records from the database, along with a brief explanation of what they mean. A FIPS 55-2 record contains a substantial amount of encoded information. The entire record is provided as part of the response, but only the most significant fields are explained by the mediator. An example is shown in Figure 4. The 132-character record has been broken up to fit on the page.

Responses are classified as "government USA census geography" followed by the post office abbreviation for the state to which the records pertain. At the time of this writing we only have the data for Virginia on line, but this will probably change. OIDs are constructed by assigning a unique serial number and a null version number to each record.

4 ALIBI and NASA

ALIBI is a general-purpose system, but it could be especially valuable to NASA if it were so applied. Some of the things that ALIBI could do for NASA are:

1. Make Information Available to the Public

   To adapt to the changing needs of the U.S., NASA has been seeking ways to make its services useful to the general public. ALIBI provides a simple interface that everyday people could use to access NASA's vast archives without having technical skill or knowledge of those archives. Mediators could be constructed for NASA databases that would respond to queries that everyday people could formulate. This task is significantly simpler than trying to construct an entire information system specifically for NASA; ALIBI already provides the software infrastructure to support public retrieval. Mediators are simple to construct and can be added in a modular fashion. Already we have a small image archive at NASA GSFC available through ALIBI.

2. Provide Unified Access to Diverse and Distributed Data

   NASA has been looking for ways to simplify access to its many different kinds of data that are stored in many different places. ALIBI could achieve this goal. Unlike other systems, ALIBI does not require the data to be translated into a standard format for them to be made available. For each different database, only the mediator needs to be customized, and this effort is minimal because of ALIBI's modular resource interface.

3. Facilitate Distribution of Centralized Databases

   NASA has a number of databases that are extremely large. It may be beneficial to break these large databases into many smaller databases that would be unified by ALIBI. ALIBI is better equipped to handle large amounts of distributed data than any distributed database, and it may be the solution to previously unsolved problems in data distribution.

4. NASA-Specific Resource Discovery

   An independent ALIBI network could be set up inside NASA to help locate and distribute internal information. This would have beneficial effects for software reuse, organizational communication, and research. However, some security enhancements will be needed to prevent private data from leaking out of NASA if the local network permits access from the outside.

5 ALIBI vs. Other Approaches

ALIBI is not the only tool capable of handling the diverse types of information that we have described, but it is the only one to support fully automatic resource discovery. Table 1 shows how ALIBI compares with other tools that are currently available. Our categories for each tool are the resource discovery mechanism it uses to find appropriate sites, the retrieval mechanism used to select and fetch individual data once a site is found, the kind of optimization used to enhance performance, the method used to determine the metadata for resource discovery, and the inherent bias of the system in terms of what kind of data it most easily handles.

In the first column, we see that ALIBI is the only system with automatic resource discovery. Archie makes it possible to search a filename index (or the *whatis* database) to try to locate something; as the prototype of all centralized catalog approaches to resource discovery, Archie defines the Archie syndrome. When centralized indexing is used to solve the resource discovery problem in a growing, decentralized system, the Archie syndrome results. The index becomes an unavoidable bottleneck. The Archie syndrome keeps being repeated because the problems with centralized indexing

Table 1: ALIBI compared with other tools

Tool	Discovery	Retrieval	Optimization	Metadata	Bias
Archie	AS	N/A (FTP)	Replication	Refresh	Filenames
WAIS	AS	FT	Replication	N/A	Text
Gopher, Veronica	Navigation, AS	Navigation, FT	Caching, Replication	Static	Hierarchical
WWW	Navigation	Navigation, FT	Caching	Static	Hypertext
ALIBI	Automatic	Flexible	Distrib. Caching	Dynamic	Atomic Data

AS = Archie Syndrome (search central index for relevant sites)
FT = Full-Text retrieval (inverted index keyword search)

only become apparent *after* the system reaches critical mass, and the exponential growth of the network is transformed into exponential index growth. WAIS incurs the Archie syndrome at the server selection stage; to find out which servers should be searched for some data, the user must either browse through a long index of servers or execute queries on a centralized catalog. Gopher supports resource discovery through navigation (the equivalent of "manual" resource discovery); Veronica adds Archie-like searches and the Archie syndrome to Gopher. The improved semantic linkage of the Web has reduced the need for centralized indices, but several of them are already available a link or two away from the NCSA home page, and Web users are taking the initiative to create more.

ALIBI also offers a new and flexible retrieval method. Archie does not compete in this arena since it is not a retrieval tool. WAIS is primarily a full-text retrieval system. Gopher and WWW support full-text retrieval at specific servers once the resource discovery step is complete; however, they also provide the ability to locate specific data through more navigation. While ALIBI does not offer navigation, its two-tiered query language supports more flexible retrieval than full-text alone. ALIBI subqueries are interpreted by resource providers in a domain-specific manner, allowing many alternatives besides traditional inverted index searching over a text-only database.

Moving on to the optimization column, both WWW and Gopher permit caching to be done at a local server or by a client program. However, this caching is not cooperative in the way that ALIBI's caching is. In the Übernet, if one site generates many queries while a nearby site is idle, that site will accumulate cached replicas of responses for the busy site. Queries from other sites that are routed through these sites may also be answered from their caches. Gopher and WWW establish direct Internet connections to archive sites, making cooperative caching impossible and causing well-known archives to become overloaded. At the time of this writing, the "home page"

for WWW at NCSA begins with "[The] NCSA Web server is overloading due to an exponential growth in connections." Replication is also used to optimize the performance of Internet tools. The centralized indices of Archie, WAIS, and Veronica have been replicated to keep up with the growing user load. ALIBI's caching effectively performs dynamic replication[13], creating more replicas of the data that are most in demand.

The metadata used for resource discovery by the various Internet tools are maintained in different ways, to different degrees. Archie updates its metadata by periodically refreshing it. Although it is costly, this keeps the metadata from becoming stale most of the time. The only WAIS metadata used for resource discovery is the index of WAIS servers, which is maintained by a central registry. The directory services embedded in Gopher and WWW presumably are refreshed like Archie, but most of the other links used for navigation are static, persisting indefinitely after the referenced data is removed and causing annoyance to users. ALIBI remains the exception, silently recovering from stale metadata in query routing tables without bothering the user and collecting metadata without incurring the cost of periodic refresh.

We do not intend to argue that stale metadata are a reason to choose ALIBI over WWW. We believe that ALIBI's fully automatic resource discovery and query-based interface are a valuable alternative to resource discovery that requires navigation. Instead of navigating to the correct site and retrieving data, or searching a catalog for the correct site and retrieving data, ALIBI users simply retrieve the data. This results in less work for the user and eliminates the network traffic generated by navigation to faraway sites. Retrieval of data from distant sites is also reduced by ALIBI since servers between them and the client (not just the endpoints) have a chance to cache responses.

Finally, there is the issue of bias. Each information system has a bias for what kind of data it most conveniently handles. It is not necessarily the case that they *cannot* handle other types of data, but it is better to be biased towards more general data types than more specific ones. Archie primarily indexes filenames. WAIS is heavily biased towards full-text retrieval. Gopherspace consists almost entirely of self-contained hierarchies at different sites with little non-hierarchical linkage, so we give it a hierarchical bias. The Web has more non-hierarchical linkage and has multimedia support with Mosaic, so we say that it has a hypertext bias. The bias of ALIBI is for atomic data (self-contained data objects). For ALIBI to derive data from fragments at many sites, a mediator must coalesce the fragments and present them to ALIBI as a single object. However, the content of the data objects and the internal organization of each information base are arbitrary.

6 Conclusions and Future Work

ALIBI provides a completely new approach to networked resource discovery and information retrieval. In this paper we have given a brief overview of ALIBI's functionality and shown how ALIBI differs from the other systems. Those readers who would like a more thorough and technical discussion of ALIBI and its components can get a copy of the dissertation "Towards a Global Federation of Heterogeneous Resources" from the University of Maryland Graduate School, Baltimore.

Future work will include producing a more complex client, further optimization of the distributed caching and classification algorithms, special handling of extremely large data objects, additional mediator development, and an e-mail interface.

A ALIBI Quick Start

Alibi.c is the client. The name of a host running a Unetd must be specified as the command line argument to the client program. In this respect it works much like `telnet`. For example:

alibi dunloggin.gsfc.nasa.gov

To try out the client, FTP alibi.c from flater/sources on speckle.ncsl.nist.gov, compile it, and run it against an existing Unetd site. A list of Unetd sites is available on request from dave@case50.ncsl.nist.gov.

The client offers the following prompt when it is time to enter a query:

```
Enter a query now, or enter 'quit' to quit.
```

A well-formed response can be an expression from the query language or the special commands **quit** and **more**. Any other response will cause a brief help screen to be printed. The **quit** command has the obvious interpretation.

Most queries will be keywords between parenthesis like this:

(cache software)

More complex queries can be built by combining keyword lists with the Boolean operators `&`, `|`, and `~`. Postfix ordering is required, and parenthesis are to be used only to delimit lists of keywords. The following query asks for data that match the keywords "cache" and "software" but not "XMS:" `(cache software)(XMS)~&`.

After a well-formed query is entered, the following message is printed:

Waiting for reply....

The client prints responses on the screen as they arrive from the Unetd. Embedded binary data are handled by prompting the user for a filename and saving the data to that file. Often a default filename will be provided that matches the name of the file at the remote site that provided the response. Unwanted blobs can be saved to /dev/null. The time needed for a reply to arrive will vary depending on the size of the response, the amount of Internet and Übernet traffic, the complexity of the query, and what the mediators did with the query. If a response is too slow, the client program can be suspended and brought back to the foreground later.

The special command **more** instructs the client to construct and submit a query to retrieve more data like the one that was just retrieved. This is done by adding a Boolean clause to the query that negates the OID of the previous response. Repeating the **more** command will add more clauses to the query until all matching data have been negated and a failure response is received.

If a query is entered that has no answer, a failure message will be returned indicating "Either there's no answer, or the answer is not available right now."

The currently available information bases are: MS-DOS software at wuarchive; rec.radio.shortwave, alt.politics.correct, *.unix.wizards; some sound files; some source code (pbmplus); some papers and the ALIBI source code distribution; some images from IBM and NASA GSFC; the entire EDGAR database from the SEC; a geographical database for all of Virginia.

Those wishing to start a new Unetd site can get the software distribution through ALIBI or from speckle.ncsl.nist.gov in flater/sources. Send e-mail to dave@case50.ncsl.nist.gov for help with installation.

References

[1] Michael F. Schwartz. The networked resource discovery project: Goals, design, and research efforts. Technical Report CU-CS-387-88, University of Colorado, Boulder, Colorado 80309, May 1988.

[2] Peter Deutsch. Resource discovery in an internet environment–the Archie approach. *Electronic Networking*, 2(1):45–51, Spring 1992.

[3] Katia Obraczka, Peter B. Danzig, and Shih-Hao Li. Internet resource discovery services. *IEEE Computer*, pages 8–22, September 1993.

[4] T. J. Berners-Lee, R. Cailliau, J-F Groff, and B. Pollermann. World-Wide Web: The information universe. *Electronic Networking: Research, Applications, and Policy*, 2(1):52–58, spring 1992.

[5] Marc Andreessen. NCSA Mosaic technical summary. Technical report, Software Development Group, National Center for Supercomputing Applications, 605 E. Springfield, Champaign IL 61820, May 1993. ftp.ncsa.uiuc.edu: /Mosaic /mosaic-papers /mosaic.ps.Z.

[6] M. McCahill. The internet gopher. In *Proceedings of the 23rd Internet Engineering Task Force*, 1992.

[7] Brewster Kahle. think.com:/public/wais/README, September 1991.

[8] David W. Flater and Yelena Yesha. An efficient management of read-only data in a distributed information system. *International Journal of Intelligent and Cooperative Information Systems*, 2(3):319–334, 1993.

[9] David W. Flater and Yelena Yesha. Managing read-only data on arbitrary networks with fully distributed caching. *International Journal of Intelligent and Cooperative Information Systems*, 1994. To appear.

[10] Gio Wiederhold. Intelligent integration of diverse information. In *Proceedings of the ISMM First International Conference on Information and Knowledge Management*, pages 1–7, Baltimore, MD, U.S.A., November 1992. The International Society for Mini and Microcomputers.

[11] Robert L. Kruse. *Data Structures and Program Design.* Prentice-Hall, second edition, 1987.

[12] Yoëlle S. Maarek, Daniel M. Berry, and Gail E. Kaiser. An information retrieval approach for automatically constructing software libraries. *IEEE Transactions on Software Engineering*, 17(8):800–813, August 1991.

[13] Ouri Wolfson and Sushil Jajodia. Distributed algorithms for dynamic replication of data. In *SIGMOD '92*, 1992.

Performance Analysis of the Unitree Central File Manager at NASA's Center for Computational Sciences Part II: New User Documentation and File Transfer Performance Enhancements

Odysseas Ioannis Pentakalos

List of Figures

This report consists of two parts. The first part briefly comments on the documentation status of two major systems at NASA's Center for Computational Sciences, specifically the Cray C98 and the Convex C3830. The second part describes the work done on improving the performance of file transfers between the Unitree Mass Storage System running on the Convex file server and the users workstations distributed over a large geographic area.

1 New User Documentation

Before we could proceed with evaluating the current performance of the systems at NASA's Center for Computational Sciences (NCCS) and enhancing the performance of data transfers between the user's workstations and the Unitree Mass Storage System we had to familiarize ourselves with the systems under consideration and with their configuration. The speed with which a new user can familiarize himself with a new environment and start to work productively depends on the quantity and quality of the documentation available at the introductory level. There are two types of documentation needed. First, the new user needs general documentation which describes the operating system or applications used at that site. This type of documentation usually is developed by the vendor of the specific product whether its an operating system or an application. Second, the user needs customized documentation developed at the local site which describes the local configuration of the computer systems, ways of accessing the systems, information about getting an account, etc. We will therefore, describe the documentation available at each of the two categories described above at this local site.

The systems that we were interested in understanding were the Cray C98 and the Convex C3830. The NCCS Cray C98 runs the Cray Research, Inc. UNICOS operating system. Its internet host name is charney.gsfc.nasa.gov and its IP address is 128.183.37.16. The Convex C3830 is the storage server and runs the ConvexOS version 11.0 which is BSD UNIX based operating system. Its internet host name is dirac.gsfc.nasa.gov and its IP address is 128.183.39.23. Since our research work focused on the mass storage system it was very important that we understood the operation of the Unitree Central Manager (UCFM) which is a hierarchical storage system manager, running as an application on dirac.

Starting from the Cray, the most useful document made available to us was the "The NCCS CRAY Y-MP User Guide, version 1.0, December 1991", written by Patricia C. Cunningham and James R. Duncan. Even though this document was relatively old and described a different Cray machine, the information in it were general enough to make an excellent

source of information. Since this information was developed at NCCS it falls under the second category described above but it includes enough general information to make it the only necessary guide for a new user. It includes the following chapters:

1. Introduction
2. Overview of NCCS computing resources: Describes the hardware and software available at each of the systems at NCCS.
3. User Services: Talks about the Technical Assistance Group, how to get in touch with them, training offered by them, etc. Also talks about other documentation available and how to order it.
4. Accessing the CRAY Y-MP: Describes the various access points to the Cray, such as the telephone lines, the network, and SprintNet.
5. Getting started in UNIX/UNICOS
6. Using UNIX shells
7. Using UNICOS compilers and loaders
8. Accessing software libraries: This is a site specific section which describes the libraries available and how to access them.
9. Using file space on the CRAY Y-MP
10. Transferring files
11. Long-term file storage
12. Running jobs
13. Maintaining and analyzing programs

14. Editing files
15. Printing files from the CRAY Y-MP
16. Scientific data visualization
17. VM/CMS vs. UNICOS
18. MVS/TSO vs. UNICOS
19. Setting your X Window environment
20. Additional documentation: This is an appendix which lists a number of references for all the material covered in this manual. It lists additional documentation for the Cray developed by the vendor, general documentation on the UNIX operating system, additional documentation for the libraries available and briefly described in Chapter 8 of the manual, and references for information on the X Window System.

The information available in this manual, both site specific but also general documentation, was enough for a new user to get him started working on the Cray machine.

The situation was not as good with the Convex machine. The only information at the time for the Convex machine were the vendor supplied manuals which could only be obtained from the vendor. On the other hand since ConvexOS is based on BSD UNIX it made it very easy to use general documentation available to get oriented with the machine. Of course site specific information such as the software available, the location of the software within the file system tree, and getting access to the system were unavailable and had to be obtained by asking people or looking around the system. This problem has now been solved by placing the information missing along with lots of other useful information on-line and accessible through the gopher server or any world-wide-web (WWW) server. The information available now through the gopher server will be described in more detail a little bit later.

Most of our work focused on the Unitree Central File Manager (UCFM) so we were more concerned about finding plenty of information on the Unitree.

UCFM is an application that runs on the Convex machine and manages the storage and flow of large numbers of files through a hierarchical storage system, consisting of various types of storage media. The most useful guide to getting started with the Unitree was the "Unitree User Guide, Convex Computer Corporation". It contains the following chapters:

1. Getting started: Describes the basics of what the Unitree does and explains a lot of the terms, such as migration, purging, and staging, that are needed to understand the rest of the manual.

2. FTP: Describes how to use the ftp command to store and retrieve files from the Unitree, as well as common errors and Unitree extensions to the standard ftp command set.

3. NFS: Describes the Network File System access path to Unitree files. Unfortunately because of performance problems this option is not available at this site.

4. Special Features: Describes the trash cans feature of Unitree which is also not available at this site.

This was an excellent guide in getting us quickly acquainted with the UCFM and its basic features. At the time there was also a lack of information about local features and problems with the Unitree as well as instructions on how to obtain an account and how to access the Unitree. This problem has also been relieved now by the development of the gopher site.

Another problem with the documentation available which has still not been solved is the lack of documentation on more technical information regarding the Unitree. Especially for people working on the performance analysis of a system, the existence of detailed documentation of the functionality of the system along with information on local site enhancements speeds up the process of producing results enormously. Examples of information that is missing are:

1. Internals information of the UCFM such as the operation of the various servers which interact during the system's operation, the data structures used to implement the UCFM, and the algorithms used within the

processes. The only guide available was a set of slides "Unitree Central File Manager: UCFM Internals Training Manual", from a training course offered by Convex. Since these were merely slides a lot of the information had to be extracted with care and doubt.

2. Information of the format used by UCFM to store files within the various storage media such as disks and tapes. Some of this information was extracted from the Internals slides mentioned in the previous slide.

3. Information and diagrams of the topology and connectivity of the storage devices at this local site. This information is extremely important to a performance analyst and must well described and kept up to date by the system's administrators or system's engineers. The information was obtained after a long process of digging the information from the system administrators.

Since this internal and local information described above is still missing but since it had to be collected by us during the performance study of the mass storage system it has been included as an appendix in this report. The information collected in this appendix must be used with care since this is a heavily used system and its configuration changes very frequently. It can be used as a starting point, on the other hand, by someone willing to extend it into a document describing the system in the necessary detail.

The information available at this time has been greatly enhanced by the use of the gopher server at sdcd.gsfc.nasa.gov or the WWW server at gopher://nccsinfo.gsfc.nasa.gov/NCCS. Most of the information needed to get a new user started is available through the menus of gopher. Also, references to additional documentation can be obtained using the gopher server. The main screen shown in figure 1 gives the availabe options when starting. Selecting menu entry number five from the menu gives the Online Documentation menu shown in figure 2. This is the central menu for all on-line documentation. Option two gives the phone number for finding out NCCS system status information. Option three describes the procedure for accessing the available systems at NCCS including both the Cray and the Convex machines. Option four, shown in figure 3, gives information about using the Cray system such as an excellent overview of the hardware and local

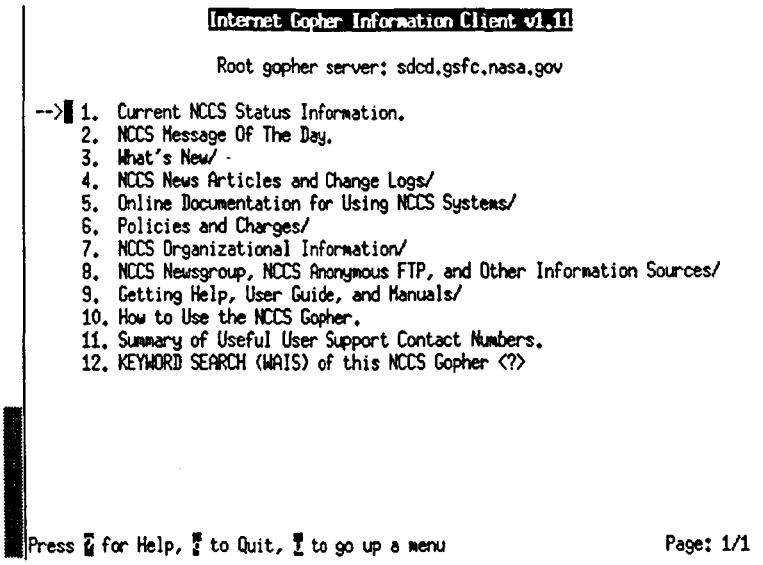


Figure 1: **Gopher Main Menu**

configuration, transferring and storing files, and references to additional documentation. Option five, shown in figure 4, has similar information to option four but for the Convex machine. Also, it includes a lot of information on using the Unitree at this specific site, information on commonly seen error messages, and additional Unitree documentation references. Finally, option six, shown in figure 5 describes some of the local software developed at NCCS for use by the users.

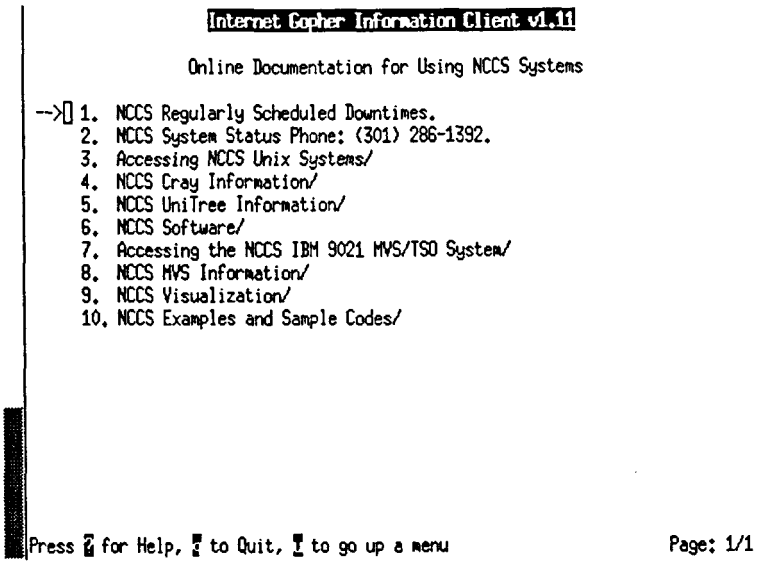


Figure 2: **Online Documentation Menu**

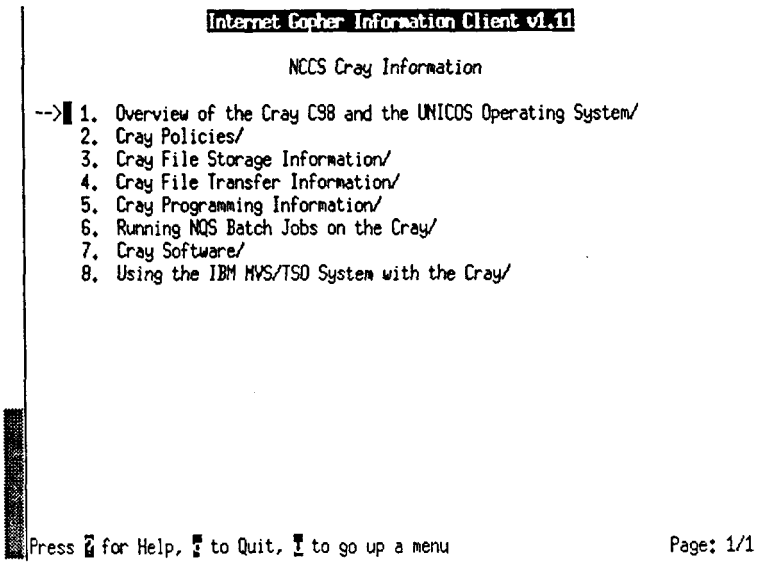


Figure 3: **Cray Information Menu**

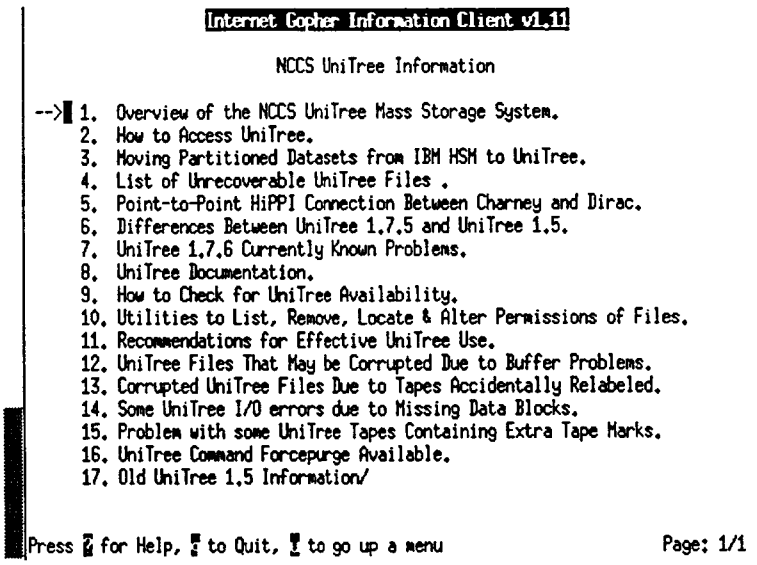


Figure 4: **Unitree Information Menu**

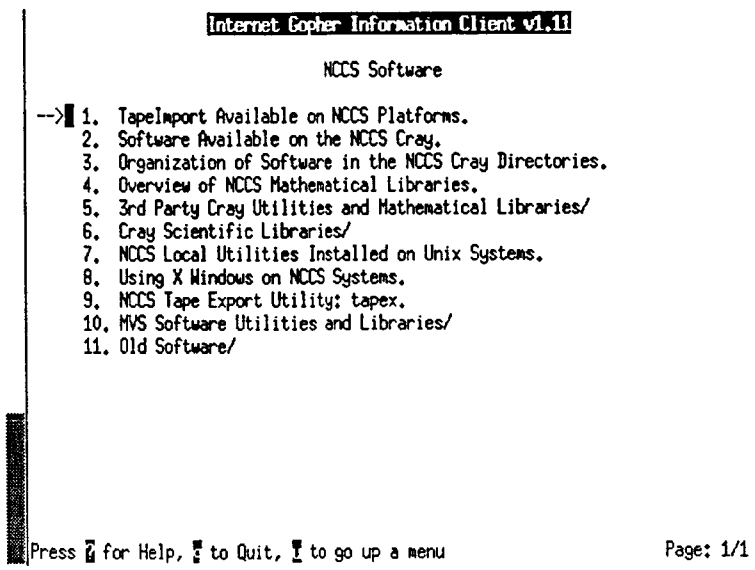


Figure 5: **Software Information Menu**

2 Introduction

Mass storage systems are used in research environments for storing data generated by scientific simulations and satelite observations in amounts on the order of terabytes. The cost of storage devices of that capacity is still very high while the rate of increase in disk space requirements by the users grows continuously. This problem is especially evident in scientific research centers where enormous amounts of data are generated on a daily basis which must be archived so that they can be analyzed at a later time [3, 6].

In this study the actual system under consideration is the Unitree Mass Storage System (UMSS) used at NASA's Center for Computational Sciences (NCCS). The original intent of this research project was to improve the performance of file transfer operations by developing a scheduler which would dynamically transfer the files a user would need for executing a computational intensive simulation at the Cray system from the Unitree mass storage system. After carefull analysis of this system we determined that the scheduler would not be a possibility since the users decide which files to use and retrieve them on their own. The idea of a dynamic scheduler would be useful in an environment where the requests for data files are indirect through NFS access to files. Due to performance and security reasons the NFS option of Unitree has not been activated or utilized at this site. In finding an alternative method of enhancing the performance of file access we noticed that the system administrators are experiencing a situation where they constantly need to purchase additional storage devices which are filled to capacity in a decreasing amount of time. The main resource whose utilization must be optimized in this case is storage capacity. Removing the redundancy in the data stored in the file system, by inserting an online compression/decompression module, is one method of increasing the effective capacity of the system without the addition of expensive hardware devices. One concern in adding online compression to a storage system is the increased load on the system since compression is a CPU intensive operation. Another concern is that if there are multiple access points to the file system significant modification must be made to the operating system to insert the compression algorithm at every write access point and the decompression algorithm at every read access point.

Since the UMSS provides access to the file system only through the ftp

clients, we inserted the data compression module within the ftp client thus dealing with both concerns mentioned in the previous paragraph. First of all, placing compression/decompression within the ftp client simplifies the installation since only the ftp clients at each workstation have to be replaced and no modifications on the UMSS software are needed. Also, since the ftp clients are distributed on each remote host of the system the additional load of compressing and decompressing files is not placed on the central mass storage system. Further, the insertion of the compression module is transparent to the user since it does not require any changes to the user interface.

The next concern was selecting a compression algorithm to use out of the multitudes that exist. The restrictions that the application places on the compression algorithm to be selected are: a) the algorithm must be online since a two-phase compression algorithm would be too slow, b) it must be able to compress at high data rates and c) it must be able to attain the highest possible compression ratio on a variety of data formats. The Ziv-Lempel and LZW algorithms were chosen for evaluation since they both satisfy the above three requirements [16, 4, 15].

In this study we examine the performance of the two online algorithms using both a sequential and a pipelined implementation of the client. The performance is compared based on the execution time of each of the algorithms and the compression ratio attained on various data formats. Two sets of files were used to test the algorithms. The first set is the Calgary Text Compression Corpus, which is a widely used benchmark for comparing compression algorithms and the second is a representative collection of files from the specific mass storage system under consideration [1]. We also examine the effect of compression on the disk cache of the mass storage system. A simulation is used to determine the effect of compressing data on the hit-ratio of the disk cache, the number of migrations of files from the disk cache to robotic storage, and the total number of bytes migrating to robotic storage. We also look at two different migration algorithms and their effect on the hit ratio and the file migrations.

Section 2 gives a brief background on data compression related terminology and a brief description of the two compression algorithms evaluated in this paper. Section 3 describes the implementations of the compression algorithms used in the evaluation as well as the method used to embed the

compression algorithm within the ftp client source code. Section 4 first describes the two distinct set of files used for evaluation of the various ftp clients developed. Then it discusses the tests performed for evaluating the performance of the various compression algorithms on the sample files. Section three describes the simulation used in this study. Section four describes the simulations performed and analyzes the results. Section five concludes the paper and discusses future work.

3 System Overview

The UMSS is a hierarchical mass storage management system which runs as a centralized application program on top of the Unix operating system and manages a hierarchical mass storage file system. The specific installation offers three levels in the storage hierarchy. Figure 6 shows the typical storage pyramid provided by most hierarchical mass storage systems. At the higher level it provides a disk array, with a total capacity of 150 GBs, which serves mainly as a cache for the lower levels. The second level has a capacity of 4.8 terabytes provided by four nearline robotic tape storage units. The third level is the offline storage vault which has the slowest transfer rate serving as the long-term repository. Users access files stored in the UMSS using the ftp protocol from their local workstations via a local area network. In addition to the ftp protocol, UMSS also provides an NFS interface to the file system but due to performance and security reasons the NFS protocol is not used by many installations including the one at NCCS. The UMSS was designed in a modular fashion in order to make possible its distribution over multiple host machines. Figure 7 shows a block diagram of the UMSS components [12].

Each of the components shown in figure 7 is represented by one or more independent daemon processes and is responsible for certain tasks. The "Name Server" resolves string file names used by the users, into unique integer identifiers, used internally by all the other components of the UMSS. The "Disk Server" keeps track of the files stored in the disk cache, providing the view of a Unix file system to the user. The "Disk Mover" is responsible for all transfers to and from the disk cache. The "Migration Server" controls the migration of files from the disk cache to lower levels in the disk hierarchy to ensure that the disk cache always has sufficient free space to operate effi-

ciently. The "Tape Server" keeps track of the files stored in the tape storage units whether online or offline. The "Tape Mover" performs all file transfers to and from a tape device. The physical device manager is responsible for managing the tape mounts, scheduling them in an order which maximizes the utilization of the system resources. Finally, the "Physical Volume Repository" is responsible for mounting and dismounting both automated online and offline storage physical volumes [14]. Any files retrieved from the UMSS are first placed in the disk cache, if they are not already there, and then are transferred to the user. Likewise, any files stored into the UMSS are first stored in the disk cache and then they are moved to a lower level of the hierarchy through migration.

4 Data Compression Concepts

In this section we define some terms which will be used in the description of the compression algorithms that follows. An alphabet is a finite set of symbols. A message is a finite sequence of symbols chosen from the alphabet. In the compression model we assume that there is a source which encodes symbols from a source alphabet. A code is a representation of messages from the source alphabet into codewords from the code alphabet.

Data compression is a coding method which reduces the redundancy in the original message during the encoding process. There are four types of codes: block-block, block-variable, variable-block, and variable-variable. A block is a message of fixed length from a given alphabet. Thus a block-block code is the encoding of the source messages into fixed length blocks, each of which is mapped into a fixed length block from the coding alphabet. Compression performance is measured using compression ratio which is defined as: $r_c = 100(S_{old} - S_{new})/S_{old}$ where S_{old} is the size of the file before compression and S_{new} is the size of the file after compression. Compression rate is the rate of compression of a file using a particular algorithm and is calculated as S_{old}/T_c where T_c is the elapsed time of compressing the file [11].

Data compression schemes, other than being classified based on the length of their encodings, are also classified based on whether they perform a static or a dynamic mapping of source strings into code strings. The encoding of source messages into codes is done using a dictionary which contains the

mapping. A static dictionary method uses a given dictionary throughout the encoding and decoding process. A static dictionary method is most often used with offline algorithms since the source messages are all given and the optimal static dictionary can be constructed before encoding starts. The dynamic dictionary or sliding dictionary method is used with online compression algorithms. The dictionary changes continuously throughout the encoding process since new strings which appear in the source messages are added while strings which have not been used recently are deleted. The main advantage of the dynamic method is that it adapts very well to changes in the temporal locality of the file being compressed [1].

The two online compression algorithms used are the Ziv-Lempel and the LZW algorithms [16, 4, 15]. Both algorithms are textual substitution based, dynamic dictionary compression algorithms. In both cases the messages from the input source are parsed into successive substrings, consisting of two parts: the *citation*, which is the longest prefix string that is in the dictionary, and the *innovation*, which is the symbol immediately following the citation. The two algorithms vary in the way they handle the innovation. In the Ziv-Lempel algorithm, each parsed substring is replaced by a pointer to the codeword in the dictionary matching the citation, and the innovation. Then the string formed by concatenating the citation and the innovation is added to the table as a new codeword. The same process is repeated until the end of the source messages by starting with the character following the current innovation.

The LZW algorithm is almost identical. There are two main differences between the two algorithms. First, after the LZW algorithm has selected the citation and the innovation of the current string, it stores them in the dictionary and then repeats the process starting from the innovation itself rather than from the character following the innovation as in the Ziv-Lempel algorithm. Also, strings are stored in the dictionary associatively using a hashing method rather than using a code index as in the Ziv-Lempel algorithm [15].

During decompression both algorithms reconstruct the dynamic table based on the compressed messages. The citation part of the code is looked-up in the dictionary and the corresponding string followed by the innovation are passed to the output. The string itself is then added to the dictionary and the process continues. The decompression algorithm for LZW compression is more complicated. By storing the codes in the dictionary during com-

pression associatively the decompression process becomes a recursive one. A more detailed discussion of the Ziv-Lempel algorithm can be found in [16] and of the LZW algorithm in [15].

5 Implementation

In order to reduce development time existing source codde was used. For an implementation of the Ziv-Lempel algorithm we used the source code of the GNU Unix utility *gzip* and for the LZW algorithm we used the source code of the *compress* utility. Both implementations deviate from the standard algorithms by including various enhancements in order to improve the performance of the utilities [5, 2].

The *gzip* utility uses a modification of the Ziv-Lempel algorithm. As described above a second occurrence of a string is replaced by a distance length to its previous occurrence and the length of the string. Distances within the file are limited to 32K bytes and lengths of strings are limited to 258 bytes. In order to improve the compression rate the algorithm also maintains two Huffman trees; in one it stores match distances and in the other literals and match lengths.

One runtime parameter of *gzip* controls the trade-off between optimizing the algorithm for speed over compression ratio. The parameter is an integer ranging from 1 to 9 where 1 attains the fastest compression for a given file and a 9 attains the highest compression ratio. The strings are stored in the dictionary using singly linked hash chains. The parameter determines at what length to truncate the hash chains as well as how to perform the matching of the strings and when to insert new strings in the dictionary [2].

The *compress* utility implements the LZW algorithm with various heuristics. Initially it starts encoding new strings using 9-bit codes and entering them in thc dictionary. Once it runs out of 9-bit codes it increases the code size to 10-bits and continues. It repeats this process up to a maximum code size, default 16, which can be set by a runtime parameter. Another modification of the LZW algorithm is that it adapts to changing blocks in the code. Once the upper limit on code bits has been reached, it periodically checks the compression ratio to make sure it keeps increasing. If the ratio

starts decreasing it clears the dictionary and starts from the beginning [5].

Both algorithms were inserted into the ftp client using both a sequential and a pipelined implementation. The sequential implementation, as shown in figure 8, when the put command is executed, passes the local file through the compression algorithm and upon completion it transmits the compressed output to the remote host. The pipelined implementation creates another process which compresses the file. The reason for implementing the compression in a pipeline is to improve the performance of the overall compression/transmission process. After a buffer of data has been compressed it is send out by the parent process while the child process starts compressing the next block of data. Figure 9 shows how the pipeline operates. Each time unit represents the time when a buffer of data has been compressed and it is send through the pipe to be transmitted.

The parent and child processes exchange data using the Unix pipe mechanism. When the ftp send command is executed, a child process is created which reads data from the file to be transmitted, compresses it and writes it to the write end of the pipe. The parent process which executes the *sendrequest()* routine in the ftp code, reads the data from the read end of the pipe and writes it out to the socket descriptor. Figure 10 shows the interaction of the child and parent processes using the pipe mechanism.

During decompression the data stream enters from the socket connection. The child process reads it and uncompresses it a buffer-full at a time, sending the output to the write end of the pipe. At the same time, the parent process reads the stream from the read end of the pipe and writes it into the local file as shown in figure 11. Table 1 summarizes the clients implemented. From this point the names of the clients shown in the table will be used in this report to describe the particular implementation under consideration.

6 Results

The four clients discussed in the previous section were tested using two sets of files. The first set of files is the Calgary Text Compression Corpus which is widely used for evaluating the performance of compression algorithms. Nine different types of text are represented in the corpus with certain types

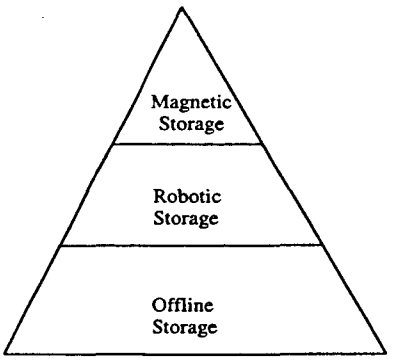


Figure 6: **Hierarchical Storage Pyramid**

Table 1: **Table of Ftp Client Implementations**

Client Name	Compression	Implementation
ftpcs	compress	sequential
ftpcp	compress	pipelined
ftpgs1	gzip -1	sequential
ftpgs6	gzip -6	sequential
ftpgs9	gzip -9	sequential
ftpgp1	gzip -1	pipelined
ftpgp6	gzip -6	pipelined
ftpgp9	gzip -9	pipelined

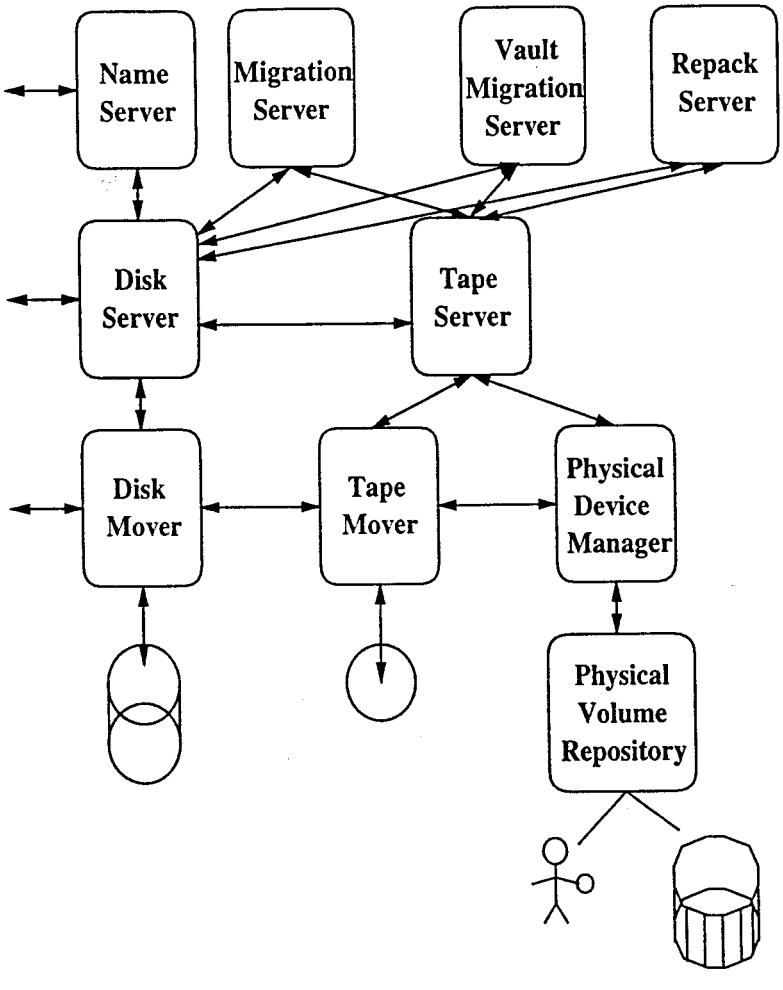


Figure 7: UMSS Block Diagram

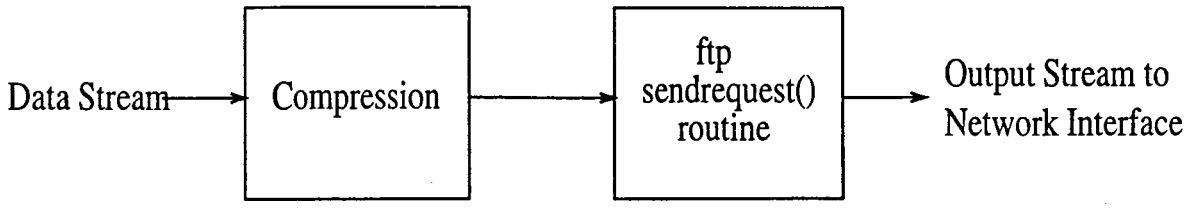


Figure 8: **Sequential Implementation**

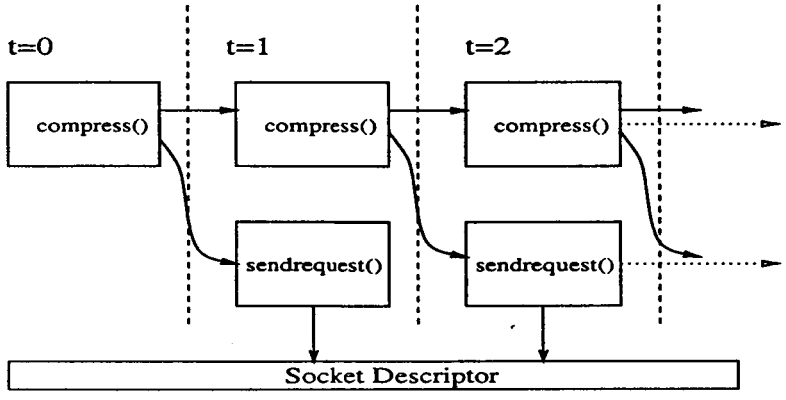


Figure 9: **The Data Compression/Data Transfer Pipeline**

having more than one representative file. Normal English text is represented using two books and six technical papers (book1, book2, paper1-paper6). A bibliography file (bib) and a few news articles (news) represent unusual English writing. Computer programming languages are represented by three files (progc, progl, progp). Non-ASCII files included are two executable code files (obj1, obj2), some geophysical data (geo) and a bitmap black and white picture (pic). A terminal session is also included (trans) [1].

The second set includes files selected from the UMSS to represent both the typical file data and file sizes found in the specific system under consideration. The files were selected as representatives from each of the research groups making use of the mass storage system. The first three files were selected from the geodynamics group (file1, file2, file3), the next two come from the Climate-Ocean-Land-Atmospheres group (file4, file5), the next two are gridded data files from the Climate Data Assimilation group (file6, file7) and the last one is an IEEE binary level 3 data file from the Climate Satelite

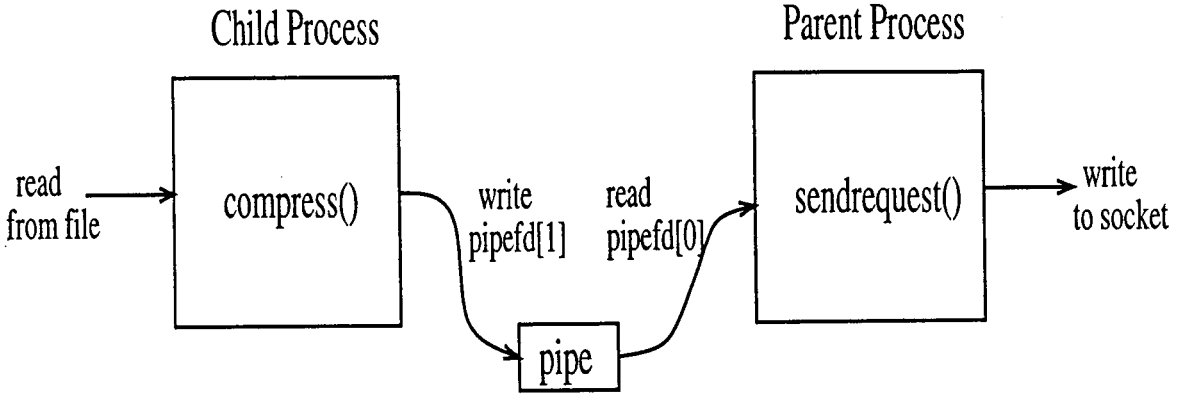


Figure 10: **Process Interaction During Compression**

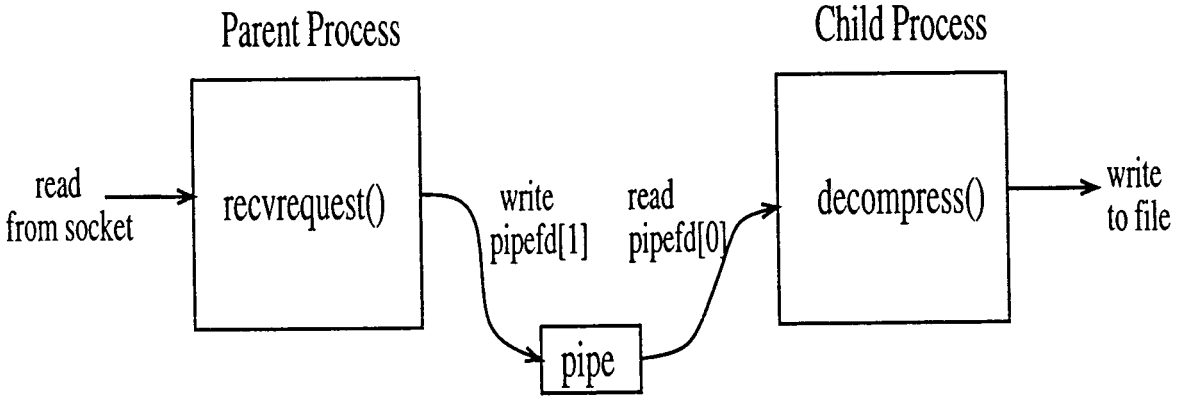


Figure 11: **Process Interaction During Decompression**

retrieval group (file8). This second set of files will be referred to from now on as the NASA Center for Computational Sciences (NCCS) files [7]. Since these files are very large and hard to process given the resources allocated to the authors they were only used for testing the compression algorithms. The transfer rate of the NCCS data files can be interpolated from the results of the files in the first set.

The first thing we looked at is how long the existing system takes to transfer the sample files from the UMSS using a plain ftp client without any compression. Table 2 shows the transfer rate and transfer time for all the sample files in the first set. This will be used as a reference point for evaluating the cost of inserting compression within the ftp client. The mean transfer rate is 30.89 Kb/sec and the weighted mean, based on the file sizes, is 30.61 Kb/sec. Of course this rate is dependent on the location of the specific host making the requests relative to the UMSS main server. To maintain consistency in the measurements, the same host was used for all measurements of transfer time and the tests were averaged over many transmission requests performed at approximately the same time of the day. Later in this section we derive two inequalities which can be used to estimate the effect of transfer time in using an online compression algorithm.

Next we evaluated the compression ratios that can be attained using the two different algorithms. The speed parameter of the gzip utility was utilized. In section 5 we described how the speed parameter is used to vary the compression algorithm of the gzip utility so that it can be optimized for higher compression ratios or higher compression rates. The value varies between 1, which attains the fastest compression rate, to 9 which attains the highest compression ratio. We tested compression using the values 1, 6 and 9. The compression algorithm will be labeled in the tables as gzip-1, gzip-6 and gzip-9 respectively. Table 3 and 4 show the compression ratios attained for each of the files in the Calgary Text Compression Corpus and the NCCS files respectively. Table 5 shows the mean compression ratio and the weighted mean attained over all files.

We then looked at the compression time of each of the algorithms. Tables 6 show the compression rates for each of the files in set 1. There are ten columns in the table. The first column lists the filename and the second column the corresponding file size. Then there are four pairs of columns

Table 2: **Ftp Transmission of the Calgary Text Compression Corpus**

Filename	File Size in bytes	Transfer Rate in Kb/sec	Transfer Time in secs
bib	111261	45.93	2.42
book1	768771	26.24	29.30
book2	610856	39.50	15.50
geo	102400	28.62	3.58
news	377109	40.07	9.41
obj1	21504	13.67	1.57
obj2	246814	25.40	9.72
paper1	53161	8.32	6.39
paper2	82199	50.58	1.63
paper3	46526	48.54	0.96
paper4	13286	31.06	0.43
paper5	11954	37.38	0.32
paper6	38105	46.88	0.81
pic	513216	19.02	27.00
progc	39611	22.73	1.74
progl	71646	12.46	5.75
progp	49379	25.76	1.92
trans	93695	33.94	2.76

Table 3: **Compression Ratios of the Calgary Text Compression Corpus**

Filename	File Size in bytes	Compress %	gzip-1 %	gzip-6 %	gzip-9 %
bib	111261	58.18	66.2	68.5	68.6
book1	768771	56.81	57.2	59.2	59.3
book2	610856	58.95	64.3	66.1	66.2
geo	102400	24.04	32.7	33.1	33.2
news	377109	51.70	60.4	61.5	61.7
obj1	21504	34.67	51.7	52.1	52.1
obj2	246814	47.87	65.2	66.9	67.1
paper1	53161	52.82	63.8	65.1	65.1
paper2	82199	56.00	62.1	63.8	63.9
paper3	46526	52.36	59.7	61.1	61.2
paper4	13286	47.63	57.7	58.5	58.5
paper5	11954	44.95	57.8	58.4	58.4
paper6	38105	50.93	64.0	65.3	65.3
pic	513216	87.88	88.6	89.0	89.7
progc	39611	51.67	65.2	66.5	66.5
progl	71646	62.10	76.0	77.3	77.4
progp	49379	61.09	76.2	77.2	77.3
trans	93695	59.18	78.2	79.7	79.8

Table 4: **Compression Ratios of the NCCS files**

Filename	File Size in bytes	Compress %	gzip-1 %	gzip-6 %	gzip-9 %
file1	69632	0.0	11.1	11.5	11.5
file2	14557184	0.0	6.9	7.3	7.3
file3	1360808	63.3	68.0	68.7	69.5
file4	193393920	11.9	32.5	33.1	33.2
file5	15612480	53.14	59.0	61.5	62.5
file6	119150208	0.0	26.5	27.2	27.3
file7	19699208	0.0	21.3	21.8	21.8
file8	199636080	0.0	21.0	21.5	21.5

Table 5: **Arithmetic and Weighted Mean of Compression Ratios**

	compress	gzip-1	gzip-6	gzip-9
Arithmetic Mean Set 1	53.3	63.7	65.0	65.1
Weighted Mean Set 1	59.6	65.7	67.2	67.4
Arithmetic Mean Set 2	16.0	30.8	31.6	31.8
Weighted Mean Set 2	5.7	26.9	27.6	27.6

for each of the compression algorithms tested. In each pair the first column is the compression time and in the second column the decompression time respectively. It is apparent by looking at these tables that compression is a much more time consuming operation than decompression. This implies that the decompression rates are much higher which means that once the files are stored in the mass storage system retrieval for processing is much faster. In a mass storage system where the number of read operations exceeds the number of write operations would definitely profit by an online compression algorithm on the clients.

Another observation we made by comparing the compression rates with their corresponding compression ratios was that using a value of 9 as a speed option with gzip will cause a small increase in compression ratio at the expense of a considerably large decrease in compression rate. Using this particular version of the compression would be suitable only for sites where increasing the effective capacity of the mass storage system is the primary concern.

The next step was to evaluate the performance of the ftp clients developed. Tables 7 and 8 list the time spent compressing and the time spent actually transferring the compressed file for the Calgary Text Compression Corpus and the NCCS files respectively for each of the four sequential clients described in the previous section. Tables 9 and 10 list the same results for the four pipelined clients. Since the time to compress the files overlaps with the time to transfer the file, only one time measurement is shown in these tables which is the elapsed time. The last column also lists the time it takes for transmitting the file without any compression for comparison. The pipelined implementation obviously provides the advantage of concurrently compressing and transmitting thus providing very good overall per-

Table 6: **Compression Times for the Calgary Text Compression Corpus**

Filename	File Size bytes	compr. secs	uncompr. secs	gzip-1 secs	gunzip secs	gzip-6 secs	gunzip secs	gzip-9 secs	gunzip secs
bib	111261	0.2	0.1	0.4	0.1	0.6	0.1	1.2	0.1
book1	768771	3.5	1.2	6.2	1.0	13.4	0.6	17.5	0.7
book2	610856	2.4	1.2	4.0	0.5	8.0	0.5	10.0	0.4
geo	102400	0.3	0.1	1.4	0.1	3.0	0.1	5.3	0.1
news	377109	1.3	0.6	2.2	0.3	3.9	0.2	5.3	0.1
obj1	21504	0.1	0.1	0.1	0.1	0.1	0.1	0.2	0.1
obj2	246814	1.0	0.3	1.9	0.2	3.1	0.2	5.5	0.2
paper1	53161	0.2	0.1	0.2	0.1	0.2	0.1	0.3	0.1
paper2	82199	0.2	0.1	0.2	0.1	0.5	0.1	0.6	0.1
paper3	46526	0.2	0.1	0.1	0.1	0.2	0.1	0.3	0.1
paper4	13286	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
paper5	11954	0.1	0.1	0.1	0.1	0.1	0.1	0.2	0.1
paper6	38105	0.2	0.1	0.1	0.1	0.2	0.1	0.2	0.1
pic	513216	0.5	0.3	1.2	0.2	2.6	0.2	32.0	0.2
progc	39611	0.1	0.1	0.1	0.1	0.2	0.1	0.2	0.1
progl	71646	0.2	0.1	0.2	0.1	0.3	0.1	0.6	0.1
progp	49379	0.2	0.1	0.1	0.1	0.2	0.1	0.4	0.1
trans	93695	0.2	0.1	0.2	0.1	0.3	0.1	0.4	0.1

formance. It appears that the most appropriate compression scheme would be the pipelined implementation of the gzip-1 algorithm. It seems to attain very good compression ratio overall while at the same time compressing at very high compression rates.

Table 7: **Compression/Transmission Times for Sequential Ftp Clients-Set 1**

Filename	ftpcs		ftpgs1		ftpgs6		ftpgs9	
	Comp.	Trans.	Comp.	Trans.	Comp.	Trans.	Comp.	Trans.
bib	0.42	1.25	0.56	1.56	1.50	2.24	2.13	0.79
book1	3.99	12.20	5.64	20.40	15.80	13.30	20.50	15.10
book2	3.11	17.70	3.81	13.20	9.94	11.70	12.20	7.44
geo	0.46	12.50	1.08	6.64	4.01	4.62	7.44	1.87
news	2.10	10.40	3.18	15.60	5.30	6.97	6.36	3.79
obj1	0.08	0.62	0.14	0.83	0.21	0.50	0.45	0.33
obj2	1.40	3.09	1.49	10.40	3.64	4.54	8.07	1.80
paper1	0.19	0.83	0.29	3.25	0.62	0.63	0.90	0.81
paper2	0.28	2.31	0.51	2.03	1.24	0.90	2.19	0.80
paper3	0.160	0.80	0.64	1.18	0.74	2.13	0.75	1.27
paper4	0.05	0.19	0.12	0.62	0.12	0.11	0.12	0.16
paper5	0.04	0.20	0.08	0.47	0.10	0.05	0.10	0.24
paper6	0.17	1.54	0.21	2.31	0.41	1.29	0.64	1.02
pic	0.96	4.05	1.88	9.01	4.00	3.51	31.00	3.54
progc	0.13	0.64	0.21	0.52	0.42	0.28	0.64	0.28
progl	0.22	0.76	0.36	0.76	0.76	1.49	2.45	0.64
progp	0.15	0.50	0.21	1.39	0.42	0.36	1.28	0.47
trans	0.35	3.06	0.42	1.11	0.74	0.70	1.17	0.57

Even though our main goal is to increase the effective capacity of the storage system it is also very important to examine the tradeoff in execution time versus increased effective capacity of the mass storage system of inserting compression at the ftp client. In other words we want to compare the amount of time taken to compress the file and send it against the current situation of simply transferring the uncompressed file. It is obvious that compression will add a factor to the cost but will the decrease in size be

Table 8: **Compression/Transmission Times for Sequential Ftp Clients-Set 2**

Filename	ftpcs		ftpgs1		ftpgs6		ftpgs9	
	Comp.	Trans.	Comp.	Trans.	Comp.	Trans.	Comp.	Trans.
file1	0.58	0.20	0.78	0.08	1.31	1.10	1.18	0.10
file2	210.00	138.00	202.00	144.00	327.00	87.10	359.00	96.80
file3	7.98	1.66	9.80	1.23	52.00	2.66	130.00	14.10
file4	1550.00	1682.00	1850.00	1530.00	2170.00	1489.00	5263.00	1463.00
file5	112.00	79.90	151.00	71.10	333.00	56.80	464.00	53.20
file6	1070.00	1113.00	1380.00	1058.00	1460.00	988.00	1789.00	969.00
file7	136.00	68.50	186.00	32.40	212.00	43.60	279.00	30.10
file8	1710.00	1727.00	2180.00	1525.00	2578.00	1437.00	4870.00	1415.00

Table 9: **Transmission Times for Pipelined Ftp Clients-Set 1**

Filename	ftpcp	ftpgp1	ftpgp6	ftpgp9	ftp
bib	1.50	2.75	3.09	2.40	2.42
book1	12.80	17.50	28.10	28.00	29.30
book2	13.2	10.40	21.70	24.00	15.50
geo	8.14	3.29	5.13	2.80	3.58
news	12.60	6.05	12.50	15.00	9.41
obj1	0.39	0.29	0.50	0.07	1.57
obj2	6.86	2.90	6.73	7.40	9.72
paper1	0.50	0.72	2.98	1.20	6.39
paper2	2.38	2.16	2.95	1.00	1.63
paper3	4.06	0.70	1.12	0.13	0.96
paper4	0.24	0.12	0.18	0.04	0.43
paper5	0.13	0.21	0.11	0.04	0.32
paper6	0.61	0.66	0.60	0.16	0.81
pic	2.08	2.98	5.67	31.00	27.00
progc	0.51	0.59	0.69	0.13	1.74
progl	0.93	0.71	1.26	2.40	5.75
progp	0.55	0.49	0.73	1.20	1.92
trans	1.13	0.94	1.45	1.50	2.76

Table 10: **Transmission Times for Pipelined Ftp Clients-Set 2**

Filename	ftpcp	ftpgp1	ftpgp6	ftpgp9	ftp
file1	0.29	1.49	1.11	0.73	1.35
file2	254.00	283.00	428.00	476.00	251.00
file3	11.20	10.60	86.00	141.00	15.60
file4	1420.00	1910.00	2360.00	6620.00	1430.00
file5	2250.00	2320.00	2710.00	5160.00	2160.00
file6	132.00	184.00	238.00	262.00	132.90
file7	1010.00	1490.00	1760.00	1920.00	980.00
file8	141.00	161.00	319.00	470.00	159.00

sufficient to gain this factor back from the file transmission ? The following inequality describes the desired relation between simply transferring the file and compressing the file before transmission.

$$\begin{aligned} \frac{S}{R_t} &> \frac{S}{R_c} + \frac{S(1-r_c)}{R_t} \\ \frac{1}{R_t} &> \frac{1}{R_c} + \frac{1-r_c}{R_t} \\ R_t &< r_c R_c \end{aligned} \tag{1}$$

where S is the size of the file, R_t is the file transfer rate, R_c is the compression rate and r_c is the compression ratio normalized to the range $[0, 1]$. The intuitive meaning of this relation is that the effective compression rate of the compression algorithm must be greater than the transfer rate of the network. If this inequality doesn't hold the online compression algorithm increases the effective capacity of the system at the expense of added time when storing the file. The above inequality applies only to the sequential implementation. Assuming that the communication time between the parent and child processes is negligible we can derive a similar relation for the pipelined implementation as shown in inequality 2.

$$\frac{S}{R_t} > \max\{\frac{S}{R_c}, \frac{S(1-r_c)}{R_t}\} \tag{2}$$

The total time of the pipeline is bounded by the maximum of each of its components. Which of the two components prevails will depend on the

particular client making the request and on the network topology. If the client is connected locally relative to the server but is a slow machine then the compression component will prevail whereas on a fast machine which is a few hops from the server the transmission component will prevail.

In addition to increasing the effective capacity of the mass storage system, placing compression on the client side has certain additional advantages. Since the files are compressed before being send to the server less traffic on the network is generated which should also increase the transfer rates. This result is apparent when looking at the faster transmission times for the sample files when comparing tables 2 and 7. Another more important advantage is the increase of the effective capacity of the disk cache. Since files are first placed on the cache when they are send to the mass storage system reducing the size of the files is the same as increasing the size of the disk cache. When the disk cache fills up a migration algorithm is used to move files to tertiary storage. Since the migration algorithm used on the UMSS is an LRU variant, the stack property of the LRU algorithm implies that increasing the capacity will definitely increase the hit ratio of the disk cache [10].

A trace-driven simulation of the disk cache was used to ascertain the effect on the hit ratio and the number of files migrated caused by file compression. The simulation used an LRU algorithm and a high-water mark for starting migration from the disk cache to tertiary storage. The size of the disk cache used was 150GB which is the actual disk cache size at the NCCS. The traces used for running the simulation were the actual ftp logs from the UMSS site. Since it would be impractical to collect the compression ratios for each of the files in the mass storage system each simulation run used a fixed compression ratio and the simulation was run for various compression ratios ranging from 0% to 70% compression. Each simulation was run using the logs from a twenty day period to remove transient effects. Then measurements were collected using the logs for the following fifteen day period. Table 11 shows a significant increase in the number of hits as the compression ratio increases.

7 Disk Cache Simulation

A trace-driven simulation of the disk cache was used to ascertain the effect on the hit ratio and on the migration of files caused by file compression and migration algorithm. A discrete event simulator was developed using the ftp request traces to drive the simulation. The disk cache size was varied from 150GB, which is the actual disk cache size at the NCCS site, to 250GB. Initially the cache was assumed to be empty. The disk cache was represented by a doubly linked list of structures which described each file entry. The information stored for each file were a unique file identifier, the file size, a timestamp of the time the file entered the disk cache, and an indicator of whether the file is stored in the disk cache or in the lower levels of the hierarchy.

Put requests were placed in the disk cache. If the file already resided in the cache or lower in the hierarchy the operation was processed as an update, ensuring that only one copy of the file existed in the entire mass storage system. For get requests, if the file existed in the disk cache then the request was considered a hit. If the file existed lower in the hierarchy it was staged in the disk cache. If the file requested did not exist in the hierarchy, it was processed as if it was in the lower levels of the hierarchy and a new entry was created for the file in the disk cache.

Migration in simulated time was performed using a high water mark as in the UCFM. If the amount of free space in the cache went below the high water mark of 75% the total disk cache capacity, files were migrated to the lower levels of the hierarchy to create more space. Two different migration algorithms were tested. The first one, was LRU based, selecting files to migrate which had resided in the cache the longest without being referenced. The second algorithm was based on the file size, migrating larger files first.

Since it would be impractical to collect the compression ratios for each of the files in the mass storage system each simulation run used a fixed compression ratio. The simulation was run for various compression ratios ranging from 0% to 60% compression.

8 Results

The ftp interactive request logs for a period of three months were used to run the simulation. The total number of references in that three month period was approximately 106,000. The references from the first two months were used for bringing the disk cache to a warm state. Then the number of hits, the hit ratio, the number of files migrating to tertiary storage, and the total number of bytes migrated were measured for fixed values of compression ratio. The simulation was run also for two different migration algorithms. The first migration algorithm, which selected files to migrate if they had resided on the disk cache the longest without being referenced, will be referred to as the *LRU based algorithm.* The second algorithm which selected files to migrate based on their file size will be referred to as the *Size based algorithm.* The hit ratio was computed as the number of hits per day over the number of get requests on that specific day.

One important observation that was made about the reference patterns used in this mass storage system was that the requests do not exhibit significant temporal locality. Users do not tend to re-use their files very frequently as in a typical file system. This implies that this specific mass storage system is used more as an archive than as a typical file system. Since the working set of the get request stream continuously changes, only low hit ratios are possible regardless of size increases to the disk cache.

In order to be able to compare the hit ratios measured with some sort of an optimal hit ratio we run the simulation on the same trace data setting the compression ratio to a value very close to zero. This allowed all the files to fit within the disk cache, imitating a disk cache of an enormous size, generating no migrations. This experiment was used to generate the optimal (OPT) disk cache hit ratios. The same method was used to compute the hit ratio of this cache as in the other cases. Table 11 summarizes the effect of compression on the number of hits for each of the experiments. The table is divided in three major column groups for each of the migration algorithms. The first column group shows the results for the LRU based migration algorithm, the second column group for the Size based migration algorithm, and the last column shows the results for the OPT disk cache. The first two column groups consist of three columns, one for each of three different compression ratios attempted. Comparing the results from the two migration algorithms against

Table 11: **Number of Cache Hits over Compression Ratio**

	LRU Based			Size Based			
r_c	0.0	0.2	0.4	0.0	0.2	0.4	OPT
1	285	285	286	285	286	286	286
2	87	87	87	104	104	104	105
3	186	186	186	186	186	186	202
4	342	342	342	343	343	343	352
5	235	241	242	435	435	435	493
6	1086	1087	1088	1089	1088	1087	1130
7	1323	1323	1323	1500	1500	1500	1698
8	143	143	143	145	145	145	153
9	60	60	60	63	63	61	63
10	250	250	250	248	248	248	252
11	321	321	321	317	317	318	324
12	422	422	422	434	434	434	464
13	371	371	371	354	355	355	409
14	376	381	381	376	376	377	436
15	1249	1249	1251	1244	1243	1244	1256

the results under OPT we see that the number of hits for both algorithms are very close to the optimal. Compression does not affect the hit ratio very much and this is because the disk cache is large enough to support the hits in the reference patterns. It should be noted that the LRU based algorithm exhibits the inclusion property as expected since the number of hits is non-decreasing with increases in the disk cache size. On the other hand, the size based algorithm in certain cases decreases with a larger effective disk cache size.

The hit ratios were also plotted in figure 12 for various compression ratios. The plot on the top shows the hit ratio variation with respect to the compression ratio for the size based migration algorithm and the bottom plot shows the variation for the LRU based migration algorithm. It is apparent from these figures that size based migration provides higher hit ratios than the LRU based algorithm. The variation in compression ratio does not have significant effect on the hit ratio and the reason for this is the same as dis-

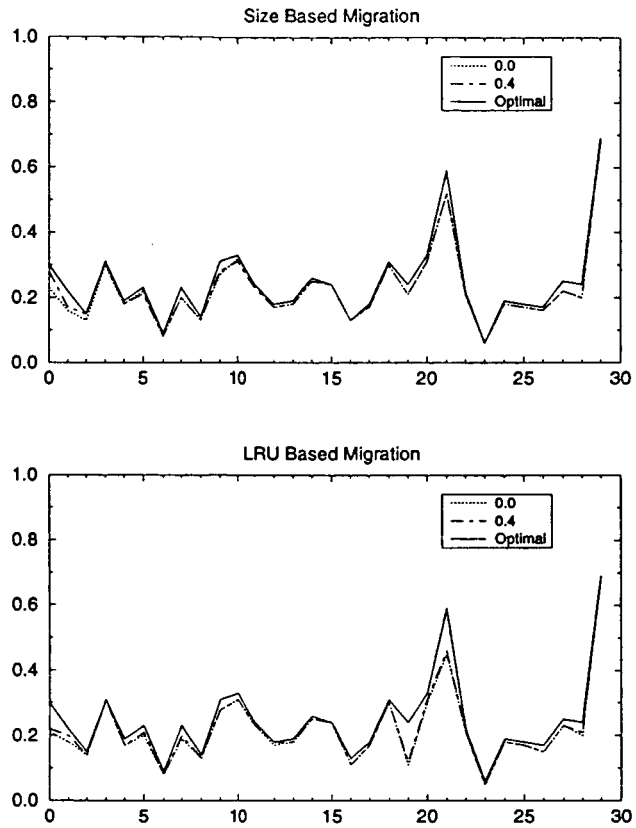


Figure 12: **Hit-Ratio versus Compression Ratio**

cussed in the previous paragraph. This implies that adding additional disks to the disk cache will not have any effect on the hit ratio based on the references analyzed. Also any further effort in improving the hit ratio by varying the migration algorithm will not generate any significant improvement on the hit ratio. The only possible method of increasing the hit ratio would be to develop a prefetching algorithm that is based on hints provided by the user.

The second part of the simulation analysis focused on the migrations. Since migration involves the use of tape drives from the robotic silos it is an expensive operation. Thus, reducing the number of migrations or the total number of bytes migrating to the tape will improve the mass storage system's performance. Figure 13 shows the number of files migrating versus compression ratio for the two migration algorithms. The LRU based algorithm maintains a consistent number of migrations and tends to smooth the

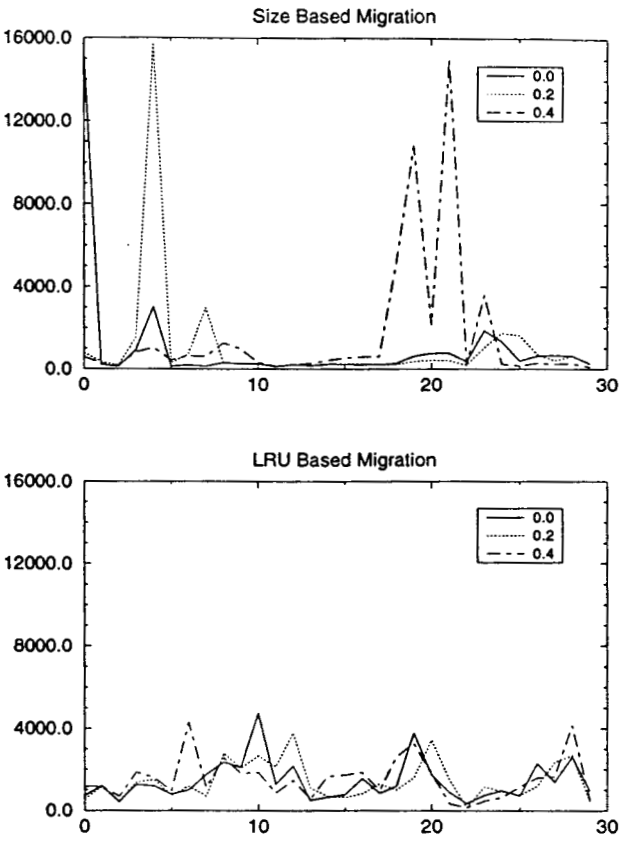


Figure 13: **Number of Migrations versus Compression Ratio**

migration operations over time. It appears that the effect of file compression is minimal. Looking at the peaks in the LRU based algorithm it appears that compression simply shifts the migration effects but does not reduce their number. The size based migration algorithm decreases significantly the number of migrations but it has the negative effect of generating on certain days tremendous migration traffic. Analyzing the file sizes for both get and put requests we found that the mean file size of files stored in the storage system is an order of magnitude larger than the mean file size of files retrieved. Since the size based algorithm removes larger files first, eventually it runs out of large files and it has to remove a huge number of small files to free space in the disk cache.

Figure 14 shows the number of bytes migrating to robotic storage for various compression ratios. It is apparent that for both migration algorithms

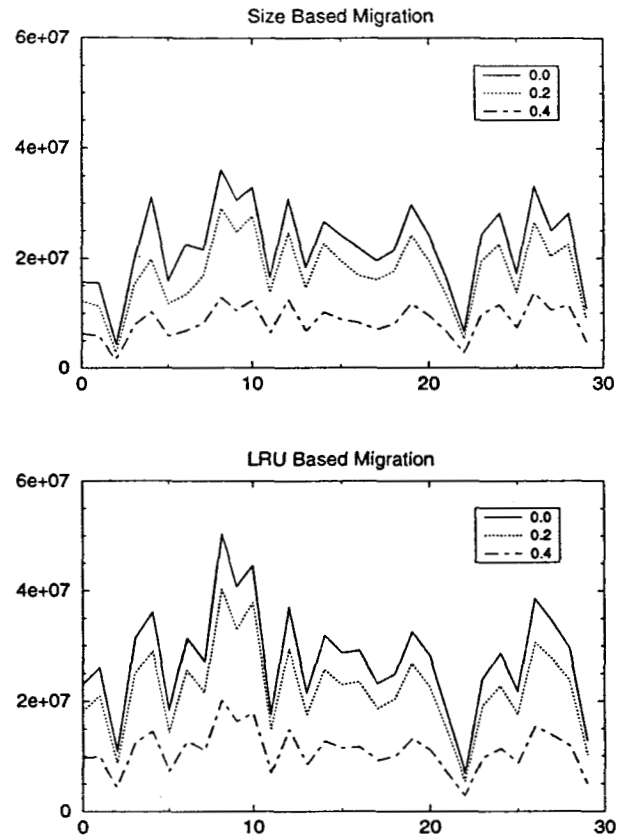


Figure 14: **Bytes Migrated versus Compression Ratio**

the higher compression ratio provides significant reduction in the number of bytes that need to migrate. The size based migration algorithm provides better performance throughout the simulation period. The time it takes the system to process a migration involves an overhead time and a data transfer time. The overhead time consists of mounting the tape on a tape drive, a seek time to place the tape drive heads at the proper location, a rewind time after the data have been written, and an unmount time. Reducing the number of migrations from the disk cache affects the overhead time while reducing the number of bytes migrating to robotic storage reduces the data transfer time.

9 Conclusion

We evaluated the performance of an online compression algorithm on the clients of a typical mass storage filesystem. Four different variations of textual substitution based, dynamic dictionary compression algorithms were implemented in both a sequential and a pipelined orientation. The eight different clients developed were tested using two sets of files. The first set was the Calgary Text Compression Corpus which was used because of its popularity as a compression performance benchmark and the second set was a representative collection of files from the NCCS.

The results showed that a good compression algorithm must be capable of compressing data at a high rate but at the same time it should be able to attain rather high compression ratios on a variety of file formats. Also, it seems that a pipelined implementation should definitely be the method to be used since it will reduce the cost of compression. From the clients tested as part of this work it appears that the pipelined clients with the gzip compression algorithm using a speed parameter of 1 and 6 are the best choice since they attain a good balance between compression rates and compression ratios.

The second part of this work involved the evaluation of the effect of online compression on the effective capacity of the disk cache. A trace driven simulation of the disk cache was used for the evaluation. The traces used to drive the simulator were collected from the ftp logs of the system. The simulation was configured to match the disk space and migration algorithm of the system at NCCS. The effect of compression was simulated by uniformly reducing the file size of the get and put requests. Various compression ratios were used in the simulation. The simulation also evaluated two different migration algorithms, specifically an LRU based and a size based algorithm.

One important observation that was made about the references at this mass storage system was that the working set continuously changes. This implies that the disk cache hit ratio cannot be improved significantly by increasing the disk cache size since get operations are usually to files that were stored in the mass storage system a very long time in the past. This effect was evident by comparing the two migration algorithms against a disk cache which was large enough to store all files stored during the three month

evaluation period. As a result both algorithms attained hit ratios very close to the optimal hit ratios of the huge cache. Comparing the two migration algorithms we found that the size based algorithm decreases the total number of bytes migrating to tertiary storage at the expense of causing occasional peaks in the number of files migrating. Both algorithms were not affected by the compression ratio due to the fact that the disk cache is of large enough size to cover the intereference pattern of the requests.

Future work will focus on evaluating various prefetching algorithms. The current simulation suggested that only the use of user hints and an appropriate prefetching algorithm can improve the hit ratio of this system. The use of transparent informed prefetching could be applied to improve the hit ratio of the disk cache by exploiting application level hints about future file accesses. Another area of future research is the implementation and evaluation of migration algorithms based on a combination of file size and cache residency time as described in [8, 9]. This simulation analysis showed that size based migration reduces the number of bytes that migrate to tertiary storage but occasionally it produces a large number of migration loads. By using a migration algorithm based on the space time product we expect that the migration peaks will disappear, while maintaining the lower number of bytes migrating.

References

[1] Timothy C. Bell, John G. Cleary, Ian H. Witten, *"Text Compression"*, Prentice Hall, Englewood Cliffs New Jersey, 1990.

[2] J. Gailly et al. Gzip, Version 1.2.4. Anonymous ftp from prep.ai.mit.edu: /pub/gnu/gzip-1.2.4.tar.gz.

[3] Randy H. Katz, Thomas E. Anderson, John K. Ousterhout, David A. Patterson, *"Robo-line Storage: Low Latency, High Capacity Storage Systems over Geographically Distributed Networks"*, Technical Report UCB/S2K-91-3, University of California, Berkeley, March 1994.

[4] Debra A. Lelewer, Daniel S. Hirschberg, *"Data Compression"*, ACM Computing Surveys, Vol. 19, No. 3, pp. 261–296, September 1987.

[5] Dave Mack, compress version 4.0. Anonymous ftp from gatekeeper.dec.com: /pub/usenet /comp.sources.misc/volume20/compress.

[6] Ethan L. Miller, Randy H. Katz, *"An Analysis of File Migration in a Unix Supercomputing Environment"*, Technical Reports UCB/CSD-92-712, University of California, Berkeley, March 1993.

[7] Ellen Salmon, NASA Goddard Space Flight Center, Private Communication.

[8] Alan Jay Smith, *"Analysis of Long Term File Reference Patterns for Application to File Migration Algorithms"*, IEEE Transactions on Software Engineering, Vol. SE-7, No. 4, pp. 403–417, July 1981.

[9] Alan Jay Smith, *"Long Term File Migration: Development and Evaluation of Algorithms"*, Communications of the ACM, Vol. 24, No. 8, pp. 521–532, August 1981.

[10] Alan Jay Smith, *"Cache Memories"*, Computing Surveys, Vol. 14, No. 3, pp. 473–530, September 1982.

[11] James A. Storer, *"Data Compression Methods and Theory"*, Computer Science Press, Rockville Maryland, 1988.

[12] Adina Tarshish, Ellen Salmon, *"The Growth of the Unitree Mass Storage System at the NASA Center for Computational Sciences"*, 3rd NASA GSFC Conference on Mass Storage Systems and Technologies, College Park, Maryland, October 19-21, 1993.

[13] James C. Tilton, Edward Seiler, *"CRUSH: A Comparative Lossless Compression Package*, to be presented at the 1994 International Geoscience and Remote Sensing Symposium, Pasadena, California, August 1994.

[14] Convex Computer Corporation, *"Unitree+ System Administration Guide, First Edition"*, Convex Press, Richardson Texas, 1993.

[15] Terry A. Welch, *"A Technique for High-Performance Data Compression"*, IEEE Computer, Vol. 17, No. 6, pp. 8–19, June 1984.

[16] J. Ziv, A. Lempel, *"A Universal Algorithm for Sequential Data Compression"*, IEEE Transactions on Information Theory, Vol. 23, No. 3, pp.337–343, May 1977.

Appendix

A The Unitree Central File Manager

The Unitree Central File Manager (UCFM) is a hierarchical distributed filesystem. UCFM is a mass storage manager which provides a transparent uniform Unix like file system to the user. The layers of the hierarchy consist of a pool of hard disks at the first layer which behave as a file cache for the overall system and robotic tape storage and free-standing tape storage at the second layer.

The data stored on the UCFM can be accessed from any local machine using either the **FTP** protocol or the **NFS** protocol. For performance reasons only the FTP protocol method is being used at NASA's Center for Computational Sciences (NCCS). When files are first transferred to the UCFM they are stored on the first layer of the hierarchy. Then, through a process called migration, a copy of each file is made available to a lower layer of the hierarchy Based on certain configurable parameters files from the highest layer are removed if they have not been accessed for a certain period of time. When the user tries to recall the file, UCFM know the highest layer location of the file and accesses it from there. Thus, files which are accesed often will be retrieved quickly whereas files which are not accessed too often will have longer access time. In a sense, the disk at the highest layer is being used as a cache for the slower lower layers of the hierarchy.

Since most of the research to be conducted will use this specific mass storage system as a base model, it is very important to understand its architecture and functionality. The description of this system will be separated into two different sections. The first section describes the hardware on which this system is running, their interconnection and configuration. This is important because any type of contention for resources at this level of the system may have a significant impact on the performance of the system at a higher level. The second section describes the software components and modules which compose the UCFM.

A.1 Hardware Architecture

This Unitree runs on a single Unix based mini-computer system as a software application. The mini-computer is a Unix based Convex C3830 multiprocessor with 3 processors. The first level of the storage hierarchy consists of 75 striped magnetics disks with a total capacity of 155 GB (gegabytes). The disks are connected to the system using 4 controllers and on each controller there can be a chain of up to 32 disks. Each controller has four ports to the 32 drives. The Unitree treats the disks as a pool of blocks.

At the next level in the storage hierarchy there are five robotic silos each with a capacity of 4.8 TB (terabytes). Each silo holds 6000 tapes. There are currently two types of tapes in the silos; the 3480 tapes which have a capacity of 200 MB each and the 3490E tapes which can have a capacity of either 800 MB or 400 MB based on the controller used to format them. Figure 15 shows the connectivity of each silo to the Convex machine. The SUN system is used for sending control signals to the silo whereas the data flows only through the control units.

Each silo has two cartridge drives and each cartridge drive can have 6 tape drives for a total of 12 drives. In this particular installation there are 6 drives per silo and the configuration is 4 drives in one cartridge drive and 2 drives in the other. Figure 16 shows the interface between the Convex machine and the silos. TLI stands for tape library interface and is an actual card that is plugged into the Convex bay and forms the interface between the VME bus of the Convex computer to the FIPS/Block Mux channel of the silo. There is a total of eight controller units that connect to the four silos and they are represented by CU in the figure. This diagram shows only the data path between the controllers and the silos. All five silos are connected by their library control unit to the SUN workstation which performs the control funtions. Library Storage Module (LSM,silo) 4 in the figure is not connected to a controller because it does not contain any tape drives. It serves the purpose of a robotic tape repository and is capable of automatically providing a specific tape stored in it to any of the other silos.

Since there are eight controllers that connect to the silos the number of concurrent tape read/write operations is limited to eight by the hardware

configuration. In order to obtain as much concurrency as possible, as is shown in the figure, the silos have been interconnected with the controllers in complex pattern. Also shown in the figure are the four free-standing tape controllers. The tape drives are identical to the ones within the robotic units and so are the tapes used. These tape drives are used for vaulting of files which means that files stored in tapes in these drives have not been accessed for a long time so a human operator will have to remove them and place them in a tape library. Because of this reason it is debatable whether this should be considered a third layer in the storage hierarchy or whether it should be part of the second layer of storage.

A.2 Software Architecture

UCFM is composed of a number of servers which manage the storage hierarchy. Each server is responsible for one specific task and thus the overall storage management task is distributed. This distribution of the responsibility and the functional separation of the components allows for load distribution, enhances the scalability of the storage system and provides more fault tolerance. Figure 17 shows a diagram of the UCFM servers and their interelation.

Figure 17 basically shows the interconnection of the servers with each other. It serves as a good overview of the structure of the UCFM but it is not detailed enough to form the basis of a queueing network model since it very general. Based on the request made by the user a job may have to visit each of the servers more than once and in a different order from what seems apparent in figure 17. A brief description of each of the servers in the figure follows.

Name Server: Its job is to maintain the Unitree filesystem structure and provide a transparent, Unix like interface, to the Mass Storage System. It resolves human-oriented names to a globally unique machine-oriented resource identifier (bifile id). It also authenticates access rights of the requestor.

Disk Server: Provides the logical means for storing and retrieving data from the disk cache. I maintains the necessary header information for mapping a bitfile id into the actual file stored on the disk.

Disk Mover: Its only purpose is to transfer file data to and from the disk cache. All requests to read and write data to and from the disk cache originate from the disk server. A response to each request is sent directly to the recipient of the file rather to the disk server.

Tape Server: The tape server performs the equivalent service to tapes that the disk server performs to the disk cache. Its objective is to maximize the use of the storage media by archiving files. It maintains all the necessary information so that it can retrieve the information back from the tapes. It receives requests from the disk server and the migration server for access to files.

Tape Mover: Its only purpose is to transfer file data to and from the tapes. It receives all its requests from the tape server.

Physical Device Manager: Its job is to manage the tape mounts and performs the mapping between bitfile ids and tape ids. It receives requests to mount tapes from the Tape Server and communicates its requests to the Physical Volume Repository to mount and dismount tapes.

Physical Volume Repository: Maintains the information abou the location of each tape and every storage device available at the tape level. It receives requests to mount tapes from the Physical Device Manager and issues mount commands to the robot or operator.

Migration Server: As its name implies it moves data from the disk cache to lower levels of storage in the hierarchy in order to increase the size of the disk cache.

The pool of disks allocated to the UCFM are separated into partitioned disks to be used by the name server and disks to be used by the disk server. The disks allocated to the name server are treated as a set of blocks. A server library, named Cachelib, maintains the 1KB blocks on all the disk partitions. It also manages an in-memory cache of 1KB blocks for optimizing the performance. The first block on each partition is used similarly to the Unix superblock to maintain the data stored in that particular partition. Cachelib manages the free list, which is a list of all the blocks not allocated to a specific file. Also, its block is sequentially numbered from 0 to $n-1$ by

the Cachelib, where n is the total number of blocks in the primary partitions, and that number is used to uniquely refer to a block.

Each partition allocated to the disk server is divided into three sections– the label, the header space and the body space. The label is 4KB long and describes the size of the disk partition, the start block for the headers and the number of headers stored on that partition. A partition can have a maximum of 2800 headers. The file header is a fixed size data structure and contains file metadata such as the number of blocks allocated to a file, the actual number of data blocks written to the file, a link count, a migration flag, the number of data fragments in the file, pointers to 8 data fragments, pointers to chains, user id of the owner, group id of the owner, access permissions and others. The body space is divided into fragments which are multiples of blocks. It will be described later how the size of a fragment is determined.

The name server manages three types of objects–directories, name space objects (files) and symbolic links. A B-tree structure is used to store the directory structure presented to the user. The rest of the objects are stored in a list. The Name Server writes a 32-byte header in the beginning of each block used for storing objects. The information stored in that header include the Node-type (root, branch, leaf), number of entries in the body and the size used by this entry. Both rood nodes and non-root nodes can be either branches or leaves. Leaves contain pairs of name/resource id identifiers and branches contain pointers to leaves or to other branches. In leave nodes along with the name/resource id pair there is a pointer to a block which contains the files metadata. That block contains a Name Server Object (NSO) header and the name of the file. If the files name is less than 16 characters long then it is stored in the directory entry, else it is stored in the block along with the metadata. The type of metadate stored include the node type (NSO), a previous NSO pointer, a block number identifier and a link count. Figure 18(a) shows an example of a directory structure and figure 18(b) shows its internal representation within the Name Server.

The disk server manages the disks allocated to it for file storage. Each such disk is formatted and has file system information on it. The disk server maintains the following data structures to keep track of the files stored in the disk cache.

Free Space Header Map This map is maintained for each disk partition allocated to the disk server. It is a bitmap of every free header block on a partition and is kept in memory.

Free Space Data Map This map is maintained for each disk partition and it is kept in memory. It is a bitmap of every data block on the partition.

Search Table This table maps resource identifiers into file header locations on disk. It is built at initialization and is kept in memory. It contains an entry for every file stored in the disk cache.

File Header Cache This is a cache of file headers in memory and is used for improving the performance of the name server and the disk server. It is indexed by the file resource identifier and is kept in memory.

When a file is created the disk server goes through the following steps. It first allocates primary header space for the new file in one of the available partitions using the Free Space Header Map. It then initializes the header fields, inserts the new header into the Search Table and the File Header Cache, creates a resource identifier for the file and returns the identifier to the creator (Name Server or Tape Server). On initial writes space is allocated in multiples of 64KB, beginning with 64KB all the way up to 4MB. This happens because network protocols don't send across the length of the file in advance. So the first request gets 64KB, the second 64KB, the third 128KB, the next 192KB and so on up to 4MB. If the file is being brought in from a lower level in the storage hierarchy then space is allocated using a first-fit algorithm in the partitions.

When a file is being retrieved from the disk cache a hashing algorithm is used on the resource id first into the File Header Cache (FHC) and then into the Search Table. The FHC is updated if the file is found in the Search Table only. If the file is neither data structures then the request is forwarded to the Tape Server.

Once the file header is located in the disk the data must be retrieved. The file header stores the fragment pointers of the file using eight fragment

pointers and four chain pointers. Figure 19 below describes the format how the fragments of a file are maintained.

The first 8 fragment pointers contain the following information: the logical partition number where the fragment is located, the logical file block number of the first data block in the fragment, the number of data blocks in the fragment and a pointer back to the partition table. A chain pointer contains the logical file block number of the first header entry, the logical partition number of the header and the position of the header within its logical block. A fragment header contains 35 fragment pointers, the number of fragment pointers in use and a pointer back to the file header. A chain header contains pointers to 47 fragment headers or pointers to 47 chain headers, the number of pointers in use and a pointer back to the file header.

The error recovery mechanism of the disk server is simple. In order to preserve the consistency of the header data structures, the tables are updated only after the data I/O completes. If either the Disk Server or the Disk Mover crash during write to the disk, an error is returned to the requestor process. Any data that was written to the disk will be ignored since the headers where not updated.

File purging is also performed by the Disk Server. Purging starts when either one of two low water marks have been reached. The one is the number of available free headers and the other is the number of available data blocks. Upon initialization, the Disk Server creates a master purge task which awakens periodically and checks the high water mark conditions. Files must have resided on the disk cache for a certain amount of time before they can be selected for purging. The criteria used for selecting files to purge are the following:

- The files dirty bit has been set after it was migrated to a lower level in the hierarchy.
- The file has resided in the disk cache for more than the minimum disk residence time (configurable parameter).
- The files purge value is large enough. The purge value of a file is Purge_val= $size * time^{exp}$ where exp is the purge exponent, $size$ is the file size and $time$ is the disk residency time. If the purge exponent

is greater than 2 then the algorithm becomes LRU and if it is less than 1 then it becomes file size based. The default value for the purge exponent is zero.

Purging stops when either one of the two high water marks have been reached. The one is the number of free headers and the other is the number of free data blocks in the disk cache. If the disk partition utilization remains below both high water marks then the purge process runs through all file headers, building a list of all migrated, non-active files and orders them according to their purge value. Once the list is built, the purge process removes each file on the list in turn from the disk cache.

Files are created in the Disk Cache when an ftp request arrives to put a file into the UCFM. Figure 20 shows the interaction between the various servers involved in storing a file into the disk cache.

The steps involved in serving such a request are as follows:

1. The ftp client sends a request to **uftpd** to store $file_1$ into the disk cache.

2. **uftpd** determines whether the file exists by trying to fetch $file_1$ from the directory.

3. **uftpd** sends a request to the Disk Server to create a new file. The Disk Server allocates spece for a new header, builds the header, inserts it in the header cache, writes it to the disk, creates the resource id for the file and creates an entry in the Search Table and the Header Cache and finally returns the resource id to the **uftpd**.

4. **uftpd** causes the Name Server to insert an entry for $file_1$ and its resource id into the working directory.

5. The Name Server sends an increment link count request to the Disk Server.

6. The ftp client sends file data to the **uftpd** for $file_1$.

7. **uftpd** sends a write/append request to the Disk Server for $file_1$ and the Disk Server allocates space on disk for file data.

8. The Disk Server sends a request to the Disk Mover to obtain data from **uftpd** and write to the allocated space on the disk.

9. Data flows to the disk through the Disk Mover.

10. The Disk Mover informs the Disk Server that the write has completed and the Disk Server updates the fragment list in the file header.

Steps 6–10 are repeated until all file data have been written.

If the operation is an ftp **get** operation the request is forwarded from the Name Server to the Disk Server. The Disk Server searches through its Header Cache and if the resource id is not found there it searches through the Search Table. If the file is found in the disk cache then the Disk Server sends a request to the Disk Mover to obtain the data from the information stored in the header and pass it to the **uftpd**. If the file is not found in the disk cache then retrieving the file involves moving the file from the tape to the disk cache and then following the same procedure described above. Figure 21 shows the sequence of operations needed to cache the file from the tape into the disk cache.

The steps are as follows:

1. Retrieve the file header from the Tape Server.

2. Once the header is received, insert it into the Disk Server's Header Cache and Search Table. The Disk Server saves the state concerning this transaction and then the connection between the Disk Server and the Tape Server is broken until the Tape Server has allocated the tape resources and the tape has been mounted.

3. Once the tape is mounted the Tape Server informs the Disk Server that its ready and the Disk Server allocates space on disk for bringing in the file.

4. The Movers are informed of what has to be done and are started.

5. The Movers exchange data in blocks until the file has been copied to the disk.

If the file happens to be on offline tape the file is cached directly to the disk as if though it was on the robotic tape storage layer. Thus caching is allowed to skip levels through the storage hierarchy wereas migration is done level by level. If the ftp request was **ls, chmod, chown, dir** then only the file header is retrieved from the Tape Server and not the entire file.

Having discussed the Disk Server lets now move to the Tape Server. In order to improve its performance the Tape Server maintains all file header information on disk partitions and all file data in tapes. The format of file data on tape is shown in figure 22

The tape label is 80 bytes and contains information identifying the tape and the time first written. The file label is 80 bytes and is not used by the UCFM. Then there are 3 unused blocks of $15\frac{1}{2}$KB each. The data blocks represent the file data fragments. Each block is $15\frac{1}{2}$KB each where the first $\frac{1}{2}$KB is a file header and the other 15KBs are file data. The file headers are used merely for positioning the tape head since the real file headers are stored on the disk. The data block size of $15\frac{1}{2}$KB and the logical file size of 128 data blocks are optimized sizes for 3480 tapes so other tapes may require different data block and logical file sizes.

A number of disk partitions are allocated to the Tape Server for storing its information. Those partitions are divided into two sets: header partitions and search table partitions. Figure 23a shows the format of header partitions and figure 23b shows the format of search table partitions.

The header partitions are further divided into primary and secondary partitions. The transaction block on both header and search table partitions holds information about writes to primary and secondary partitions. When the Tape Server comes up after crashing it reads through the transaction blocks to check the consistency of the major data structures. The free space header map is a bitmap of the available space on the header partitions. Headers are then stored in groups of 11 per 4KB block. The interesting fields of the primary file header are:

- Number of data blocks, in bits, written to the file.
- The Unix link count, the owners UID and the owners GID.
- Access permissions.

- Number of data fragments in the file.
- Pointers to three data fragments.
- A pointer to a fragment table and two pointers to chain headers.
- Identifies the header as the first primary or duplicate primary header

The fragment header holds pointers to 13 fragments, and the chain header holds pointers to 26 chain pointers. Chain pointers point to fragment headers. Fragment pointers point to data fragments on tape and include the address of the fragment on tape and the fragment size. The Search Table maps file resource identifiers into file header locations on the disk. The table is kept in the search table partitions and contains an entry for every file in the UCFM that has migrated to the tape at least once. The Tape Map maps allocated body space on tape. For each tape it maintains the tape identifier and the block count which is the number of currently used blocks on the tape. The Book Sector journals modifications to file headers. The Write Record logs labels of tapes currently being written and the last block written. This allows the migration server to begin a migration round at the block following the last block written at the conclusion of the preceding migration round. The Tape Server also maintains the File Header Cache which, as its name implies, caches in main memory the most recently accessed file headers indexed by the resource identifier.

When the Tape Server receives a request for retrieval or storage of a file it retrieves the header information, determines the tape identifier of where the file is stored and sends the request to the Physical Device Manager (PDM). The PDM maintains two tables in memory: the Array of Tape Devices (ATD) and the Queue of Mount Requests (QMR). The ATD lists for every request the tape identifier, the status of the device (available/not available) and the mode of the request (read/write). The QMR lists for every request the device identifier where the tape will be mounted, the tape identifier which will be mounted and the mode of the request. When the PDM receives the request, it queues it and forwards it to the Physical Volume Repository (PVR).

The PVR maintains a Queue of Mount Requests (QMR) and the Location Table. The QMR stores for each request the device identifier, the tape

identifier and the mode of the request. The location table in this specific implementation is located on the Sun workstation as shown in figure 15. The PVR forwards the mount request to the Sun. The Location Table maintains information about the exact location of each tape whether it is in the silo or on a shelf. Figure 24 shows a trace of a mount request through the various servers.

The steps for servicing a mount/dismount request are as follows:

1. The Tape Server sends a request to the PDM to mount a particular tape on any drive. The PDM places the mount request on its QMR.

2. The PDM forwards the mount request to the PVR. The PVR places the request on its QMR and determines whether the tape is on-line or off-line storage. If it is on-line the PVR selects the drive for the mount.

3. The PVR issues a mount request to either an on-line mounting mechanism (i.e., robot) or an off-line mounting mechanism (i.e., operator).

4. The PDM via the Tape Mover, polls all tape drives to determine when and where a tape is mounted. When the mount completes the PDM updates its ATD and dequeues the mount request from its mount queue.

5. The PDM sends to the PVR a request to dequeue the mount request from its QMR when the mount has completed.

6. The PDM replies to the Tape Server when the mount completes with the tape drive id where the tape is mounted.

7. To servide a dismount requestt he Tape Server sends a dismount request to the PDM for a particular tape. The PDM gets the tape drive identifier of where the tape is mounted from its ATD.

8. The PDM issues a rewind/unload command to the tape drive via the Tape Mover.

9. The PDM sends the dismount request to the PVR, identifying the tape drive where the tape is mounted.

10. The PVR issues a dismount request if the tape is mounted in an on-line facility.

The Migration Server is responsible for migrating files from the disk cache to robotic tape storage. It maintains a single table in memory called the Migration Table (MT). The MT is indexed by file resource identifier and for each migratable file it lists the file identifier, the size of the file, the time of last header modification (ctime), the time of last read (atime) and the time of last body modification (mtime). The Migration Server recieves a list of migratable files from the Disk Server including the three time statistics for the file (ctime, atime, mtime). Figure 25 shows a trace of a migration server run. The steps are as follows:

1. The Migration Server obtains a migratable file list from the Disk Server. The Migration Server the rumbles through the headers to determine whether there are enough files to start migration
2. The Migration Server sends a migrate file request to the Tape Server.
3. The Tape Server obtains the file header from the Disk Server, allocates space on the disk for the header if the file is new and allocates space on tape for the data if the file size is greater than zero. It then writes the header to the disk.
4. The Tape Server then sends a mount request to the PDM for the tape containing the allocated data blocks.
5. The PDM determines the correct tape and forwards the mount request to the PVR.
6. The PVR sends the mount request to the mounting mechanism and the tape is mounted.
7. The PDM senses the tape is in the drive via the Tape Mover and replies to the Tape Server with the device identifier of where the tape is mounted.
8. The Tape Server sends the request to the Disk Server.
9. The Disk Server forwards the read request to the Disk Mover.
10. The Tape Server sends a write request to the Tape Mover.

11. Data starts to flow from the disk to the socket and then to tape via the Movers. The Tape Server writes the final header and deallocates any old fragments in the Tape Map.

12. The Tape Server replies that its done to the migration Server.

13. The Migration Server sends a mark migratable file request to the Disk Server and the Disk Server marks the file as migrated if it has not been modified since the migratable file list had been generated.

Steps 8 through the first part of step 11 are repeated until all the data for each file are read. Steps 2-13 are repeated for each file in the migratable file list.

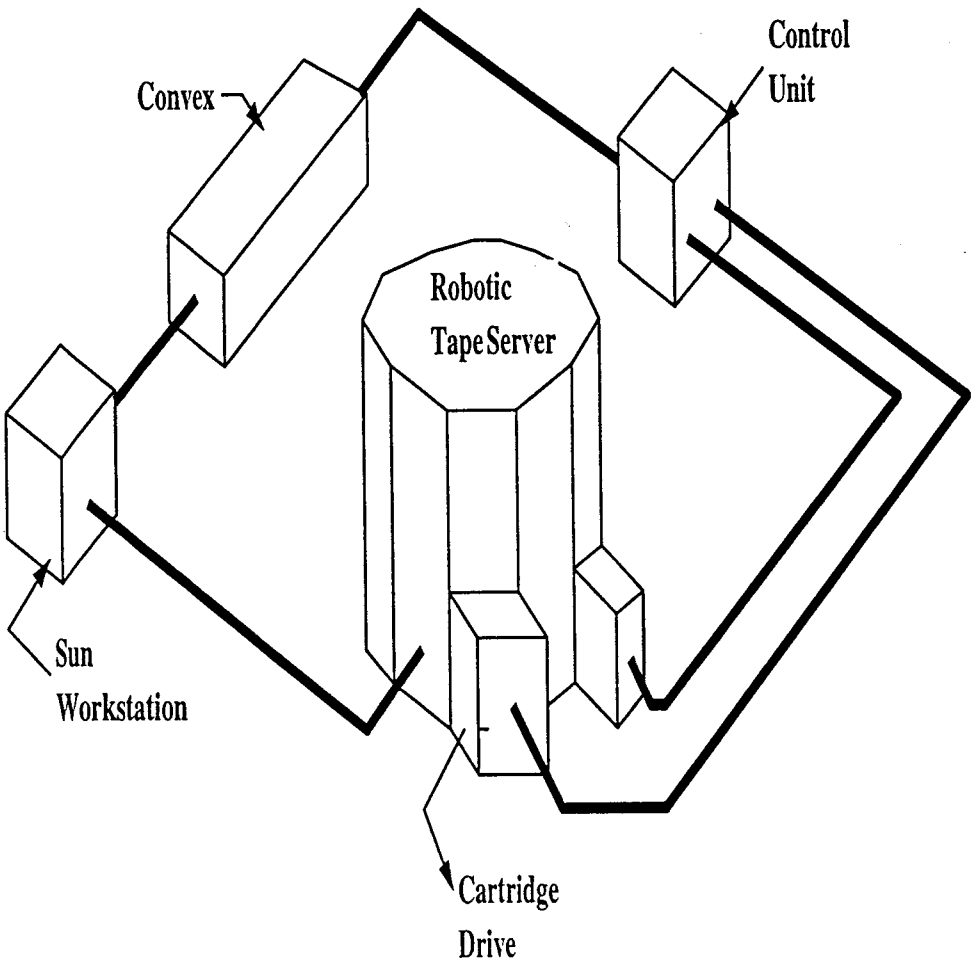


Figure 15: Silo/Sun/Convex Connectivity

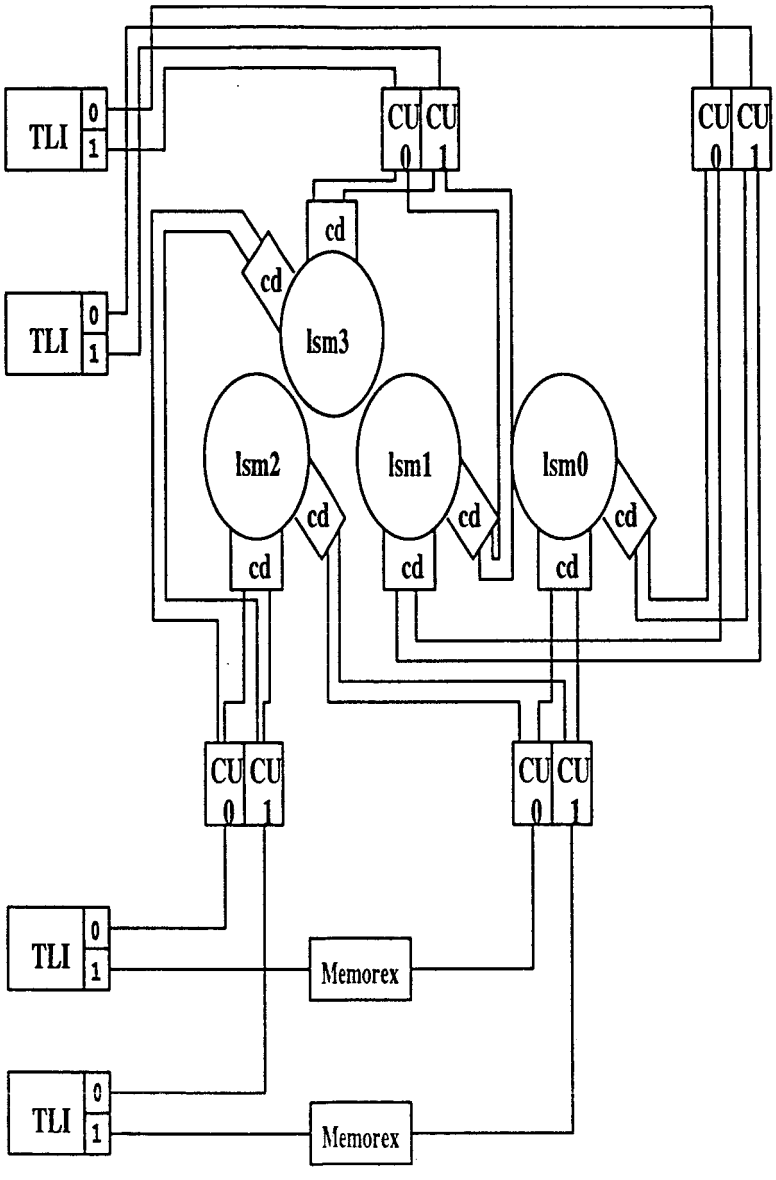


Figure 16: Interface between Silos and the Convex

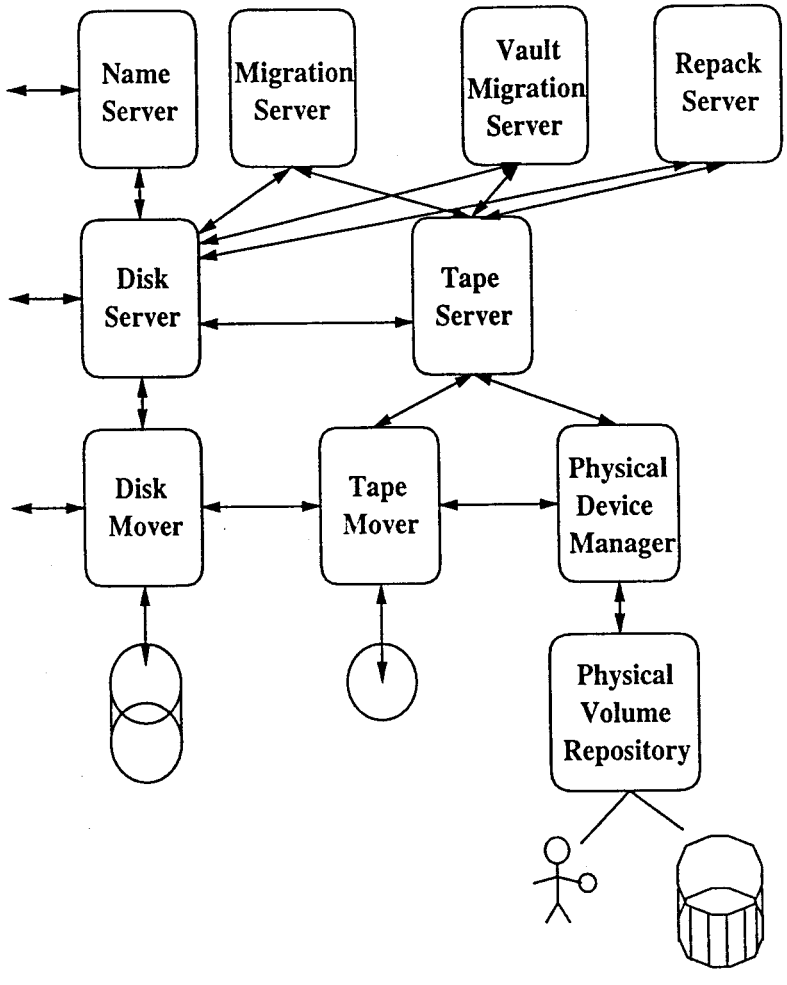


Figure 17: UCFM System Architecture

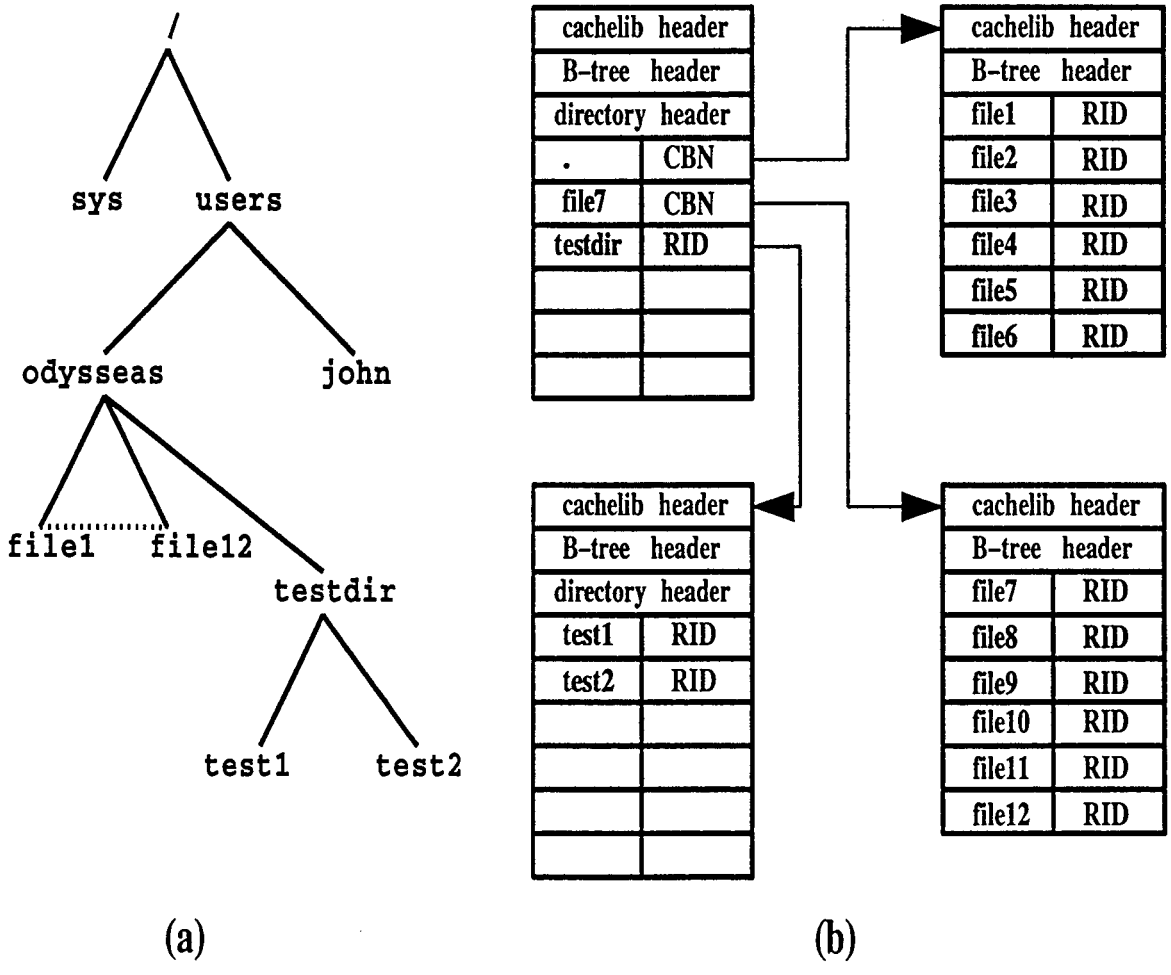


Figure 18: Directory Structure Representation by the Name Server

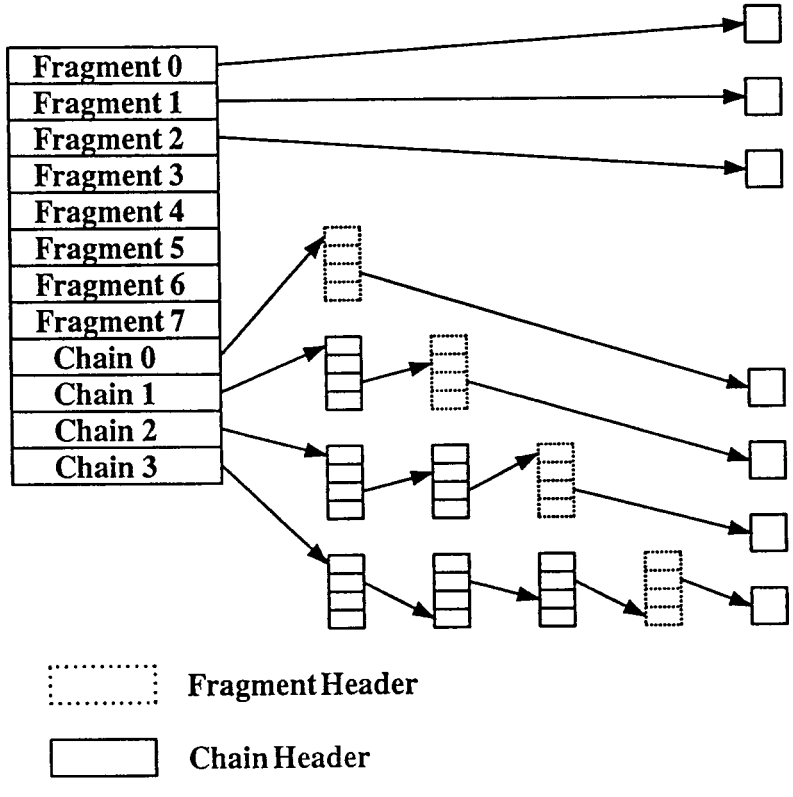


Figure 19: **Fragment and Chain Pointers in File Header**

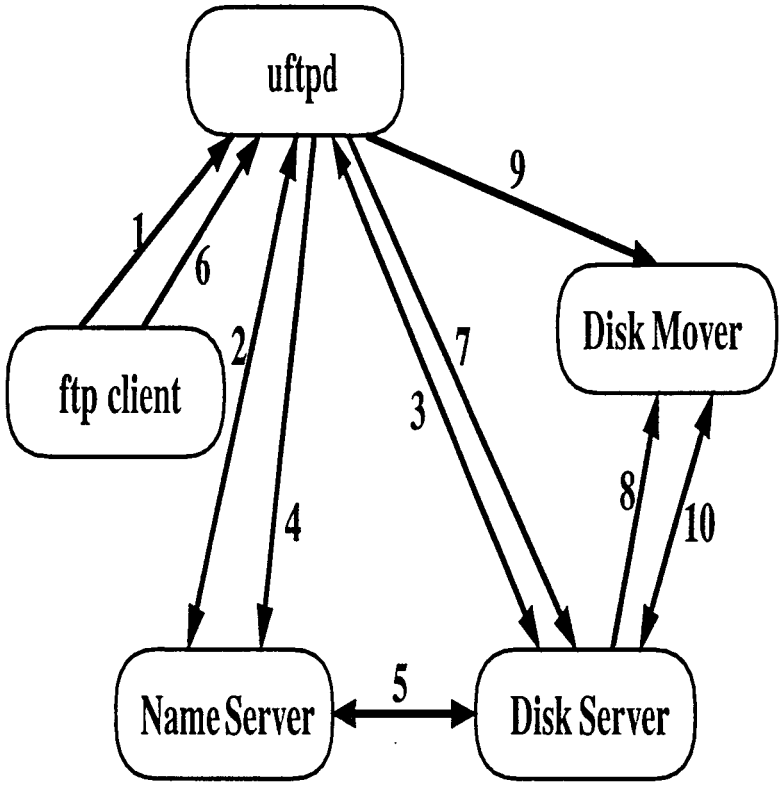


Figure 20: **Sequence of Operations in Serving an ftp put Command**

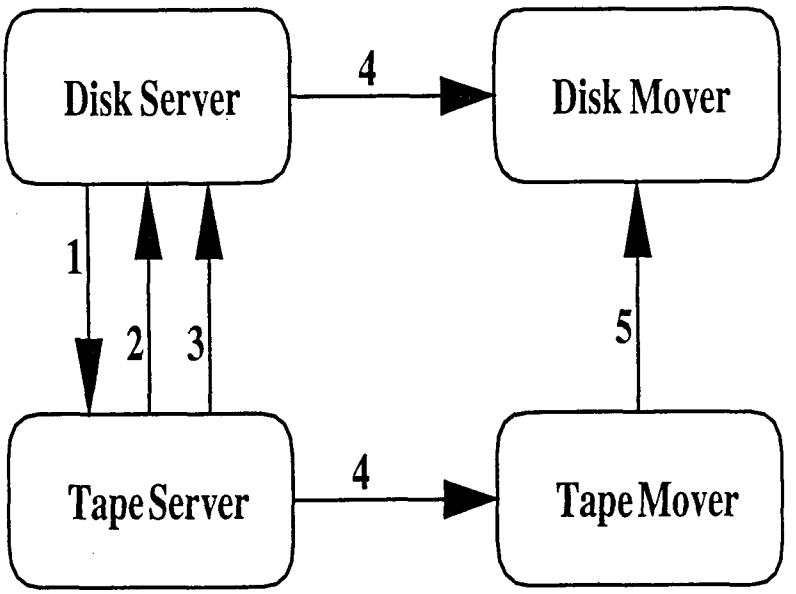


Figure 21: Sequence of Operations in Caching a File

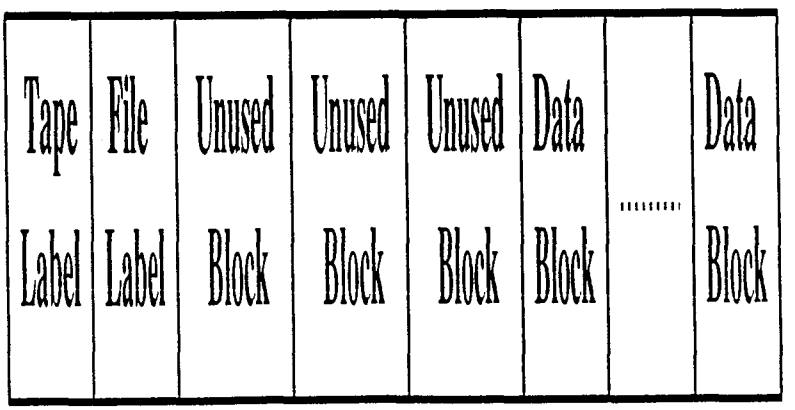
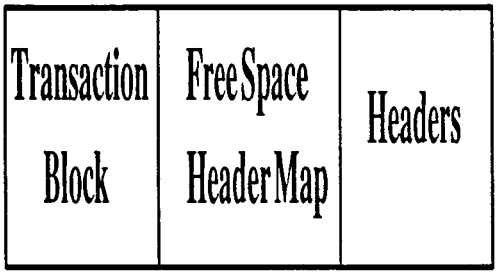
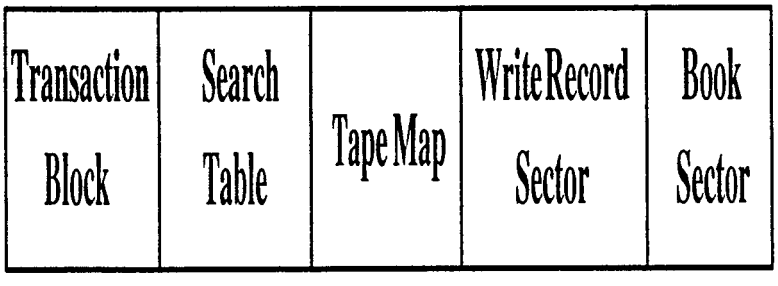


Figure 22: Format of File Data on Tape



(a)



(b)

Figure 23: **Format of the Disk Partitions of the Tape Server**

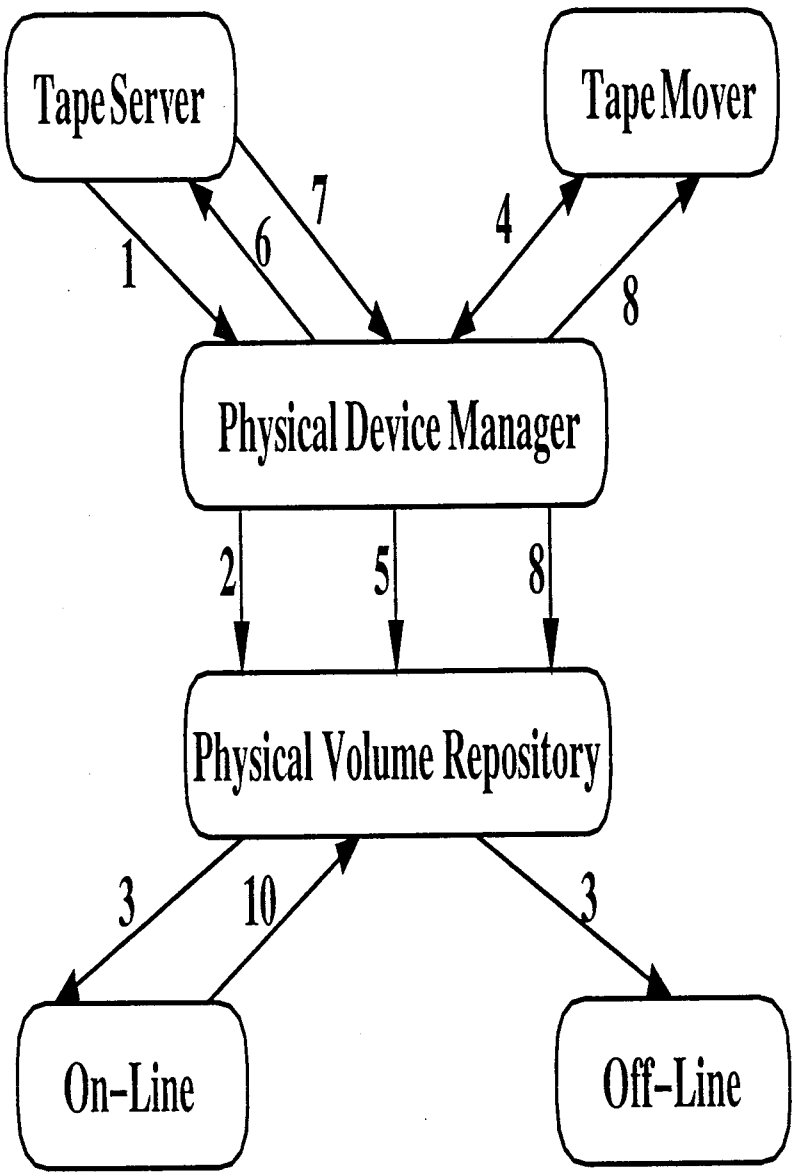


Figure 24: Sequence of Operations in Serving a Mount/Dismount Request

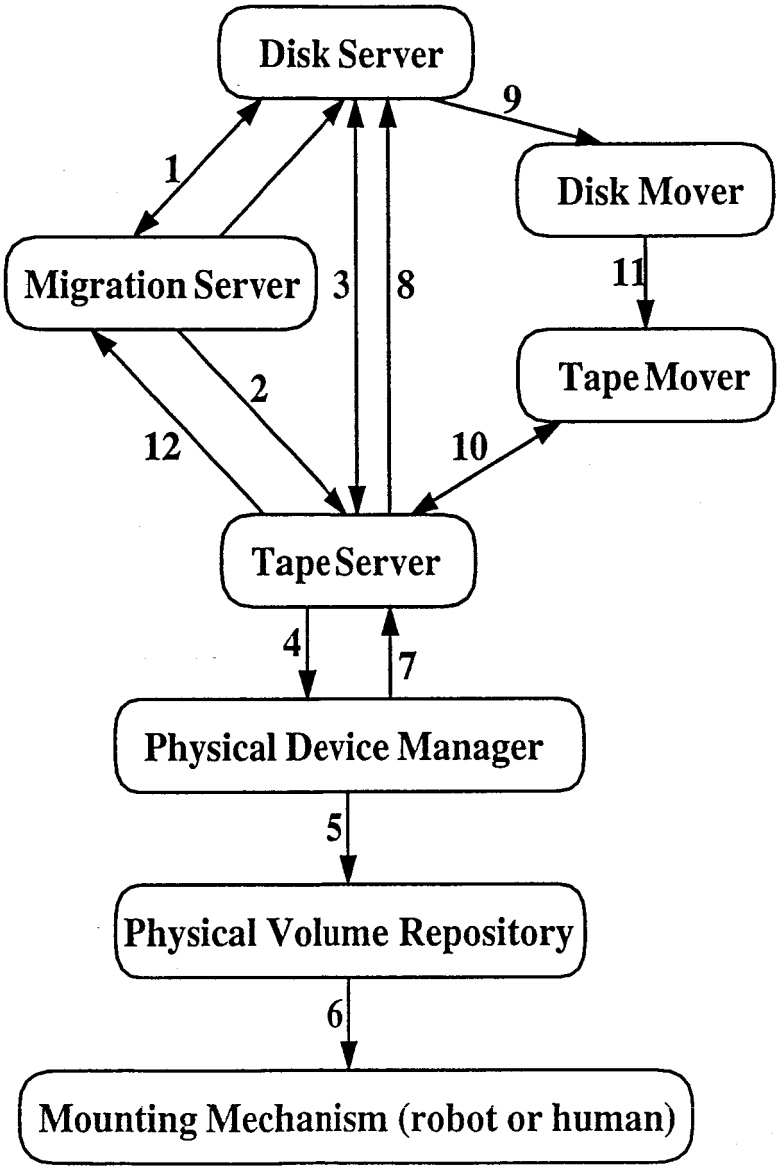


Figure 25: Sequence of Operations of a Migration Run