TrISH – TRANSPARENT INTEGRATED STORAGE HIERARCHY: A HIERARCHICAL STORAGE MANAGEMENT SYSTEM FOR THE 4.3BSD UNIX OPERATING SYSTEM

by

Sidney G. Bytheway

A thesis submitted to the faculty of
The University of Utah
in fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

The University of Utah

March 1997

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a thesis submitted by

Sidney G. Bytheway

This thesis has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

Chair: Robert R. Kessler

Gary Lindstrom

Lee A. Hollar

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the thesis of Sidney G. Bytheway in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date

Robert R. Kessler
Chair, Supervisory Committee

Approved for the Major Department

Dr. Tom Henderson
Chair/Dean

Approved for the Graduate Council

B. Gale Dick
Dean of The Graduate School

ABSTRACT

Online disk space is a valuable, relatively expensive, and frequently scarce resource that is often abused by users who squander it on large quantities of inactive data. Large inactive files should instead be moved to cheaper and more abundantly available offline or near-line storage. Users, however, are often reluctant to utilize offline storage because it is difficult to use. An extension to the UNIX operating system that transparently migrates inactive data between online and offline storage is examined, enhanced, and evaluated.

To my wife,

Trish

and my children,
Benjamin, Allie, Aaron and Brooke

who were my support throughout this project.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENTS

I would like to thank Jay Lepreau for his time and guiding hand through this project and for providing financial support and a machine for this work to be developed and tested on, Mike Hibler for his technical expertise and patience with "dumb" questions, and the rest of the Computer Science Laboratory staff for their help and patience at critical times. The following registered trademarks occur in the text.

CHAPTER 1

INTRODUCTION

Over the last few years, we have seen an enormous increase in the storage capacity of computer disk drives. An average disk drive a few years ago held 10 to 40 megabytes of data and cost about $25.00 per megabyte. Today disk drives hold up to 9,000 megabytes of data and cost about $.30 per megabyte.

1.1 Disk Storage

Amazingly enough, as the amount of available disk space has grown, so has the amount of data that needs to be stored. The space requirements of graphical images, scientific data, and even business data have kept pace with, and even outstripped, advances in disk drive technologies. The need for more, inexpensive storage space is as much of a need today as it was 10 years ago. A study conducted for Epoch Peripheral Strategies found that storage increased on average of 60% to 80% per year. "The usage of disk space is going up faster than the price of disk is going down"[10, p. 46].

An informal survey[8] of internet sites revealed some surprising facts about disk space usage. This survey asked sites to voluntarily run a program that gathered data about the sizes of files on their systems. Data for over 1000 file systems containing 12 million files with 250 gigabytes of data were gathered. The data are summarized into Table 1.1 and graphed in Figure 1.1

By examining the graph in Figure 1.1, it becomes evident that 90% of the files are less than 16K in size and that the remaining 10% (which are larger than 16K) consume 90% of the disk space. Thus, 10% of the files consume 90% of the disk space.

Table 1.1. UNIX File Sizes

File Size (Max. Bytes)	*Number of Files*	*% of Files*	*Cumm % of Files*	*Disk Space in Megabytes*	*% of Space*	*Cumm % of Space*
0	147479	1.2	1.2	0.0	0.0	0.0
1	3288	0.0	1.2	0.0	0.0	0.0
2	5740	0.0	1.3	0.0	0.0	0.0
4	10234	0.1	1.4	0.0	0.0	0.0
8	21217	0.2	1.5	0.1	0.0	0.0
16	67144	0.6	2.1	0.9	0.0	0.0
32	231970	1.9	4.0	5.8	0.0	0.0
64	282079	2.3	6.3	14.3	0.0	0.0
128	278731	2.3	8.6	26.1	0.0	0.0
256	512897	4.2	12.9	95.1	0.0	0.1
512	1284617	10.6	23.5	566.7	0.2	0.3
1024	1808526	14.9	38.4	1442.8	0.6	0.8
2048	2397908	19.8	58.1	3554.1	1.4	2.2
4096	1717869	14.2	72.3	4966.8	1.9	4.1
8192	1144688	9.4	81.7	6646.6	2.6	6.7
16384	865126	7.1	88.9	10114.5	3.9	10.6
32768	574651	4.7	93.6	13420.4	5.2	15.8
65536	348280	2.9	96.5	16162.6	6.2	22.0
131072	194864	1.6	98.1	18079.7	7.0	29.0
262144	112967	0.9	99.0	21055.8	8.1	37.1
524288	58644	0.5	99.5	21523.9	8.3	45.4
1048576	32286	0.3	99.8	23652.5	9.1	54.5
2097152	16140	0.1	99.9	23230.4	9.0	63.5
4194304	7221	0.1	100.0	20850.3	8.0	71.5
8388608	2475	0.0	100.0	14042.0	5.4	77.0
16777216	991	0.0	100.0	11378.8	4.4	81.3
33554432	479	0.0	100.0	11456.1	4.4	85.8
67108864	258	0.0	100.0	12555.9	4.8	90.6
134217728	61	0.0	100.0	5633.3	2.2	92.8
268435456	29	0.0	100.0	5649.2	2.2	95.0
536870912	12	0.0	100.0	4419.1	1.7	96.7
1073741824	7	0.0	100.0	5004.5	1.9	98.6
2147483648	3	0.0	100.0	3620.8	1.4	100.0

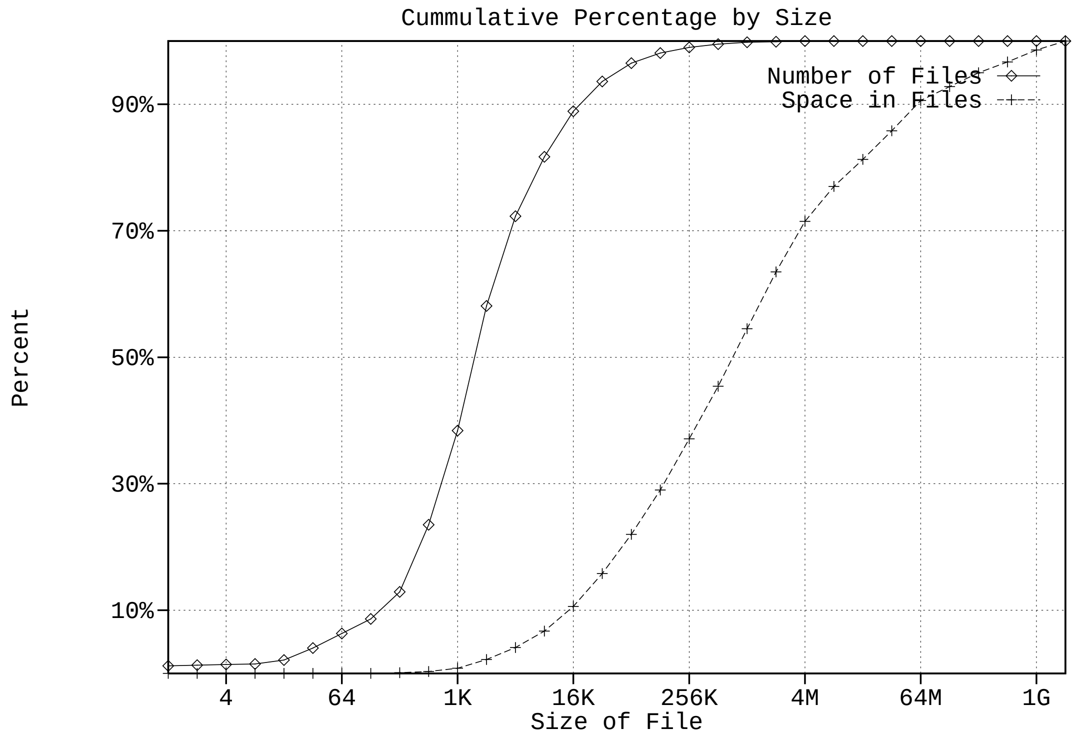

Figure 1.1. UNIX File Size Graph

Similar results were obtained at the National Center for Atmospheric Research (NCAR). An analysis of NCAR's mass storage system (MSS) showed that about half of the files contained 98% of the data[11]. Given that their MSS limited file sizes to 200 MB and that files larger than this were broken up into multiple 200 MB files, a storage system that would allow larger sized files would have an even smaller ratio of files to data.

NCAR also concluded that "only 5% of all files are referenced more than 10 times, 50% of the files in the trace were never read at all, and another 25% were read only once"[11, p. 429]. Additionally "for files which were referenced, the second reference came soon after the first"[11, p. 429], and "about one third of all requests came within eight hours of another request for the same file"[11, p. 430]. They concluded that "files can be migrated to a less costly storage medium if they are unreferenced for only a few days"[11, p. 431].

These conclusions clearly indicate that if the relatively few, very large, inactive files could be moved to less expensive storage, fewer disk drives would be needed. Even a 50% reduction in disk drive needs would save an organization a considerable amount of money.

1.2 Offline Storage

The storage capacity of offline storage devices such as tape drives and optical disk drives has also been increasing at a tremendous pace over the last few years. The old 1/4–inch tape drives of a few years ago held less than 250 megabytes of data. Today helical scan tapes, such as 8mm and 4mm tape drives, can hold up to 10,000 megabytes of data [1]. The cost for 8mm offline storage space is less than 1 cent per megabyte [1], far less than even the cheapest disk drive.

Sadly, these offline storage devices are deficient in two major areas: they are slow and inconvenient to use. The data transfer rate of a disk drive is around 10 megabytes per second, whereas the transfer rate of an 8mm tape drive is less than 1/2 megabyte per second. Human intervention is often required when accessing offline storage. For instance, when a user wants to access data on a tape, the tape must be manually loaded into the tape drive. Because UNIX does not provide a way to automatically keep track of what data are on which tape, the user is left with this tedious and error prone task. Additionally, since filesystems are not built on tapes, the access to data on tapes is only through UNIX utilities such as *tar*. Because of this, data on a tape are not directly accessible by a program. It must first be extracted from the tape before it can be used. The risk of losing data on a tape is high, especially when writing data to a tape that already has data on it. It is no wonder that few users actually use offline storage.

Near-line storage devices are a relatively new technology. A near-line device consists of an offline device, such as a tape drive or optical disk drive with removable media and a robot mechanism that can insert and remove media from the drive without human intervention. Tape and optical disk juke-boxes are examples of near-line storage devices. These devices do not require human intervention, but they are difficult to use directly, because the user is still left with the responsibility of keeping track of what data are on which media. For the purposes of this paper, when referring to offline devices, both near-line and offline devices are implied unless explicitly stated otherwise.

1.3 Storage Hierarchy

A *Storage Hierarchy* (see Figure 1.2) can be constructed out of online, near-line, and offline storage. Speed and cost increase going up the hierarchy and storage size increases going down the hierarchy. At the top of the hierarchy is online disk storage. It is expensive and fast and has limited availability. In the middle of the hierarchy is near-line storage. It is moderately inexpensive and somewhat slow when compared to online storage. At the bottom of the hierarchy is offline storage. It is very inexpensive, abundantly available, and slow.

As evident in the file size analysis discussed earlier, if the few large files in the file system could be moved through the storage hierarchy to near-line or offline storage, less of the expensive online storage space would be required to support more data. The problem is making near-line and offline storage easily usable.

All users of the UNIX operating system are familiar with the filesystem. The UNIX filesystem has a hierarchical directory tree structure where users can organize

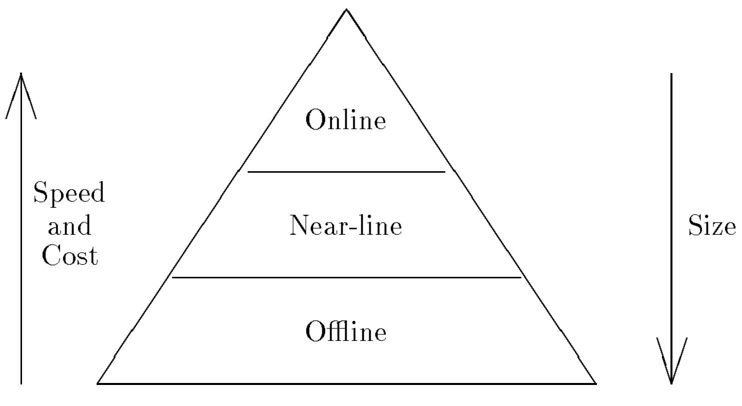

Figure 1.2. Storage Hierarchy

their data into files and groups of files. Since most file manipulation utilities and user programs only access data that are in the UNIX filesystem, the user must (manually) move any data that are on offline storage to the filesystem before processing it. If offline storage could be transparently integrated into the UNIX filesystem, users could begin using offline storage without changing the way they access and manage their data.

1.4 Hierarchical Storage Management

A system that enables the automatic movement of data through the storage hierarchy is the topic of this thesis. The system is named TrISH, short for Transparent Integrated Storage Hierarchy. TrISH is integrated into the UNIX file system and transparently manages near-line and offline storage as an extension to online storage space.

TrISH is based on BUMP, the BRL/USNA Migration Project[12]. The BUMP system was extensively modified, extended, and enhanced. The base set of features and enhancements that were to be added to the TrISH system include the following:

- Add NFS server capabilities.
- Refine migration system architecture.
- Enhance migration system performance.
- Improve interactive access to migrated files by reducing the access latency.
- Develop support to reduce "out migrations" by reusing valid offline copies of files that have been previously migrated.
- Develop better offline device handling.
- Add support for robot mounted (*near-line*) devices.
- Enhance migration system databases.
- Add support for the UNIX quota system.

- Develop system administrator migration tools.
- Develop user migration tools.

As the system was designed, other features were also added. These are discussed in Chapter 3.

1.5 Thesis Outline

A complete description of the BUMP system is provided in Chapter 2. At the end of that chapter, I draw some conclusions about BUMP, its strengths, its weaknesses, and its possible future enhancements. In Chapter 3, I explain how the TrISH system improves on the BUMP system. I compare and contrast the two systems and explain, in general terms, the goals behind the improvements made in the TrISH system.

A detailed description of TrISH is contained in Chapter 4, including a general operational description, extensions made to the operating system and to a few of the UNIX system utilities to support TrISH. I also discuss in detail the TrISH supporting programs and daemons. In Chapter 5, I analyze the specific features of TrISH and draw some conclusions about how successful TrISH is at addressing the problems raised in Chapter 2. A quick survey of a few hierarchical storage management systems, including a comparison of them to TrISH, is presented in Chapter 6.

The appendixes are full of grungy and boring details for those interested in reading about the modifications to the operating system, the structure of messages between different components of TrISH, abstracted interface functions, and the TrISH configuration file.

CHAPTER 2

THE BUMP MIGRATION SYSTEM

The US Army Ballistics Research Laboratory and the US Naval Academy developed a system they named *The BRL/USNA Migration Project*, or "BUMP" for short. The BUMP system is the starting point for the work done on TrISH.

2.1 Background and Goals

The BUMP system was originally designed to address two specific issues. First they wanted to utilize online disk space efficiently, and second they tried to eliminate errors due to full filesystems. There were a number of goals defined for the system to address these issues.

There were two primary design goals of the BUMP project. The first goal was to develop a UNIX-based file migration system that would cause a filesystem to appear as though it had much more storage than the device (disk drive) on which it was created. The second goal was to do it transparently so that no unmodified programs would be able to tell the difference between a migrated file and a regular file, except for possible delays in the completion of the `open()` system call[12].

To accomplish these two primary goals, the following specific goals were used as guidelines in designing the BUMP system:

- Separate the migration *policy* from the migration *mechanism* so that a site can change the policy without having to also change the mechanism.

- Keep modification of the UNIX kernel to a minimum. Implement in user-level code as much of the migration system functionality as possible.

- Preserve the size of the on-disk inode structure. This would allow easier implementation of the system in existing sites and keep system utility modifications to a minimum.

- Provide the ability to support a variety of secondary storage devices without changing the internal structure of the migration system.
- Provide robust system operation, even under adverse conditions such as a heavily loaded system or operating system crashes. The reliability and availability of a system running BUMP should be similar to a system not running BUMP.
- Allow multiple copies of data on offline storage to enhance reliability.
- Provide the capability of having data online and offline at the same time so that space reclamation or in-migration can be performed quickly.
- Support different types of secondary storage devices by providing access methods for moving data between them.

It is also important to acknowledge the "consciously chosen limitations" of the BUMP system. These limitations included the following:

- Only regular files would be considered for migration. No directories or special files would be migrated.
- File migration services would be provided only to the machine connected to the disk system. Remote file access through NFS would not be supported.
- No support was to be provided for creating files that are larger than the filesystem in which they were created.

2.2 How BUMP Accomplished Its Goals

The BUMP system provides facilities to migrate data transparently between the standard UNIX file system and offline storage. It consists of some UNIX kernel modifications to create and support migrated files, a daemon that communicates with the kernel and performs migration system tasks for the kernel, a few UNIX system utility modifications, and a set of tools to manage migrated files.

2.2.1 Migrating Files to Offline Storage

In order to migrate a file, a number of things need to happen. The file must be identified as one that can be migrated. The algorithm used to determine if a file can be migrated compares the product of the file size and number of days since last accessed with a system administrator defined *badness* value. If the file has a badness value larger than the filesystem's limit, the file is eligible to be migrated.

All eligible files are *premigrated*. This is the process of transferring all of a file's data block pointers (the filesystem data structures that link a files data to the file) to another file, assigning the file a unique number, called the *file handle* and stamping it with the machine's identifier. The file and its offline data are linked using the *file handle* and the *machine id*. The inode (file) that was the target of the data block pointer copy is called the *staging* file. It exists in a special directory in the filesystem called the *staging area*. The database is updated to show that the premigrated files have copies of their data in the staging area.

Premigrated files are then copied to offline storage, and the database is updated to show that the files have copies of their data on offline storage. An important detail is that the data are actually read from the staging area files.

2.2.2 Out of Space Error Handling

When the free space in a filesystem drops below a predetermined limit, the kernel notifies the migration daemon. The migration daemon forks a process which releases the storage held by *all* premigrated files; that is, all of the data in the staging area are released. If filesystem free space never falls below the limit, premigrated files retain their online data.

A process that is writing data when a filesystem becomes full will block until the migration system can create more free space. If no free space can be created, the blocked process may wait forever.

2.2.3 Reloading Migrated Files

When a process attempts to open a migrated file, it is blocked in the open() system call while the migration daemon restores the file's data. If the file cannot be restored, the open system call returns an *open error* to the calling process.

When a migrated file is accessed, the migration daemon is notified by the kernel. The migration daemon forks a reload process which has the responsibility of reloading the file's data. The reload process consults the database to determine the offline media that will be used to restore the data. The data are copied from offline storage to a staging area file. The data block pointers from the staging area inode are transferred to the migrated file's inode. The migrated file's inode type is changed from *migrated* to *regular* and the staging area file is deleted. Of course if the staging area data were never released, no access to offline storage is needed.

2.2.4 UNIX Process Flags

A process can set a few different *process flags* to modify the behavior of the migration system. One process flag causes the system to return an error when opening migrated files. Another causes the migration system to reload files in the *background* after returning an open error. The last process flag will cause the system to return an *out-of-space* error when writing to a full filesystem rather than blocking the process. The default is to block the processes when it opens a migrated files and when it attempts to write to a full filesystem.

2.2.5 Kernel and Migration Daemon Communication

A virtual communication link, implemented by a migration system device driver, is used by the UNIX kernel and migration daemon to send and receive messages. Over this link the kernel sends out-of-space and reload requests to the daemon, and the daemon sends back the status of reload requests. This communication link is also used by the BUMP system calls to request services and information from the kernel.

2.2.6 UNIX Kernel Modifications

To support migrated files the following changes were made to the UNIX kernel:

- A new *migrated* inode type was created
- Two fields were added to the on-disk inode. They store the file handle and machine id for migrated files.
- A system call to migrate a file (transfer its data block pointers to a file in the staging area and update the file handle and machine id inode fields) was added.
- A system call to unmigrate a file (transfer data blocks from the staging area file back to the original file) was added.
- Process flags to support customization of the migration system were added.
- System calls to query and set the new process flags were added.
- A virtual communication device driver was added to allow the kernel and the migration daemon to send and receive messages from each other and to support the new BUMP system calls.

The following kernel routines were modified to support migrated files in the following ways:

ufs_open() Notifies the migration daemon when a migrated file is opened. The requesting process is blocked (unless the nonblocking process flag is set) until the file has been successfully restored.

dirremove() Informs the migration daemon when migrated files are deleted. The migration daemon invalidates the database entries of deleted files.

ufs_create() Notifies the migration daemon when a migrated file is re-created so it can invalidate the database entries for the file.

ufs_setattr() Sends a message to the daemon when a migrated file is being truncated to length zero so it can invalidate the database entries. If a migrated file is being truncated to a length other than zero the file is restored as usual before the requesting process may continue.

execve() Checks that migrated files are restored correctly before attempting to execute them.

newproc() Passes the migration process flags to child processes.

ufs_getattr() Changes the status of migrated files to regular files so that the requesting process does not know that the file has been migrated.

alloc() Notifies the migration daemon when filesystem free space drops below the allowed minimum.

realloccg() Blocks the requesting process when the filesystem is out of space until free space has been created by the daemon.

2.2.7 UNIX System Utility Modifications

In general the user does not need to know if a file is migrated, but there are circumstances in which the user will want or need this information. The following system tools have been modified to recognize and work with migrated files and, where appropriate, provide file migration information to the user.

Changes to `fsck` When fsck encounters an inode type it does not recognize, it resets it to a regular inode. Since fsck does not normally know about migrated inodes, it was modified to recognize and perform integrity checks on migrated inodes.

Changes to `ls` When a long listing of a migrated file is requested, the file type flag is set to 'm', indicating a migrated file. The file handle and machine id fields are listed along with the file name, providing a way to identify the file handles and machine ids of migrated files.

Changes to `find` Another file type (-type m, indicating a migrated file) was added to the find utility since migrated files do *not* test true as regular files (-type f). Also the -ls option was modified to display an 'm' for migrated files, similar to the *ls* command.

Changes to `restore` When the restore program encounters a migrated file, it creates a new (regular) file, ensures that the inode entries are valid and converts this regular file into a migrated file. (The `dump` program was not modified since it does not interpret the inode fields.)

2.2.8 BUMP System Tools

The tools to manage migrated files include:

- Nightly jobs that migrate files from the filesystem to offline storage
- Programs that restore files from offline storage to online storage
- Scripts to reclaim online storage when free space in a filesystem drops to an unacceptable level.

2.2.9 BUMP Databases

The BUMP system was built to be portable and self-contained; it could not assume there would be any particular database system available at all sites. Consequently a database system was built into it. The data are stored in ASCII text files with newlines separating records. Normal UNIX text processing commands may be used to process and create reports from the database files.

The two permanent databases in the BUMP system are the file handle database and the volume database. The file handle database relates file handles to offline storage media. It is used to locate a file's data on offline storage. The volume database stores the device type, location, size and other information about offline volumes.

2.2.10 System Administrator and User Tools

The BUMP system suffers from a chronic lack of system administrator tools. The only tools provided by the system are the out-migration and out-of-space utilities. There are no system administrator planning tools.

There are no user tools to force a file to migrate, nor are there any tools to "batch" reload migrated files. There is no way to release manually the staging area space of files that will not be needed again.

2.3 Analysis of BUMP

2.3.1 Full Filesystem Errors

BUMP adequately addresses the issue of eliminating out of space errors in the filesystem. When the kernel notifies the migration daemon that disk space is low, the daemon is usually able to create free space. When it cannot, the reason is frequently that there are large active files still in the filesystem.

Since the BUMP system creates free space by simply deleting data in the staging area, free space can be created quickly. Often, however, more free space is created than is necessary, since *all* premigrated files in the staging area are deleted. A system administrator has to carefully balance the number of files that are premigrated. A large number of premigrated files allows for fast free space creation, but the penalty is more files that must be restored from slow offline storage.

A better approach would be to have high and low free space values, called the free space *high* and *low water marks*. When free space drops below the low water mark, staging area files are deleted until the free space again reaches the high water mark. In addition, if large numbers of files are premigrated, the free space creation process has a deep reservoir of quickly obtainable free space. The system administrator should be able to adjust the high and low water marks to balance free space with free space creation processing.

2.3.2 Offline Device Support

The BUMP system successfully provides the ability to add support for many offline devices. When support for a new offline device is needed, new device access method functions are written to mount and unmount the media, read and write a block of data, write file labels, etc.

BUMP does not, however, provide the ability to effectively support various *types* of devices, like online and *near-line* devices. The device abstraction should be at a higher level, rather than its current low-level implementation. For instance, all BUMP routines assume that media must be mounted before it can be opened, but online devices are never unmounted, and near-line devices may already be mounted. If the abstraction level were to be raised to a higher level, (for instance at the opening, closing, reading and writing level), then the device access methods would be able to provide more efficient handling of the specific device and its media.

An unfortunate side effect of the way in which BUMP deals with offline storage is the fact that the offline media are mounted and unmounted for each reload request. If a number of files all reside on the same offline tape and these files are to be reloaded, the BUMP system would mount and unmount the tape for each file. A better approach would be to restore all files from the tape while it is mounted.

Offline devices must be dedicated to the BUMP reload processes excluding them from being used by other processes and even from use by other BUMP processes. The offline devices should be able to be *moved* between processes. A device manager would be the best way to handle detachable devices.

Much of the offline storage management functions, like tape and device management, have been built into the BUMP system. If a site already has a tape management system, it is unusable by the BUMP system. The offline management functions should be abstracted out of the BUMP system.

2.3.3 BUMP's Database System

The BUMP system had an implied goal of not being dependent on any particular third party database system. This is an admirable goal, but there are a number of problems with the way it was implemented. First, the database was built into the migration system, rather than being a separate entity. Consequently the database interfaces were also built into the system instead of being abstracted out. This makes it impossible to add support for a site's favorite database system without rewriting most of BUMP. Secondly, the database provided by BUMP is very slow and inefficient. It has been determined that database updates are in fact one of the worst bottlenecks in the system.

A well-defined interface between the BUMP system and the database would allow a site to use its favorite database. Of course a database system should be provided with the system just in case a site does not have one.

2.3.4 Data Integrity and Reliability

Data integrity is guaranteed by the offline device access methods. These software routines generate and add CRC codes to the data as they are written to the offline device. This may be in addition to any CRC and ECC codes generated by the device itself. The CRC codes are validated as the data are read from the offline device, and bad data are discarded.

By allowing multiple copies of a file to exist on different offline media and device types, the system may shield itself from single point failures. It also enhances this reliability by dividing a file into data chunks called *granules*. Data are written to and read from offline storage in granules. If a granule is unreadable, another copy of the granule can be used instead. Some near-line devices, however, have better reliability than the online storage from which the data were migrated. On these devices the overhead of granule management is inappropriate.

Granules are also used to deal with files that are larger than the offline storage media. If, while writing a granule, the end of the volume is reached, the data written up to that point are discarded and a new copy of the granule is written to a new volume.

The data integrity and reliability features of BUMP are beneficial and should be retained. However, they should be implemented in the device access methods rather than in the BUMP system routines. If it is appropriate for a device to use granules to improve reliability, then the method can implement them. If a style of media management other than granules is more appropriate, then the device's method can provide it. BUMP forces CRC checking for all offline copies. However, many offline devices perform CRC checks already. This additional CRC checking adds overhead without improving upon the reliability or integrity of the system. Options like CRC checking should be optionally implemented in the device access methods.

2.3.5 Efficiency and Usability

As a user searches through a directory looking for a particular file, many files are accessed. If a significant number of the files in the directory are migrated, many of them will inevitably be reloaded. Most of these files are not needed by the user and will probably not be modified. In the BUMP system these reloaded files have to go through the migration process all over again by being copied to offline storage. If offline copies of unmodified files could instead be reused, the efficiency of the migration system would be greatly improved. Additionally if a feature could be designed and implemented such that these "false reloads," could be eliminated, the interactive feel of the system would improve dramatically.

When BUMP restores a file, the entire file is restored to the staging area and then moved into the original file. The requesting process cannot access any of the data in the file until the entire file has been restored. If instead the data were made available to the requesting process as soon as they was read from offline storage, the available data could be processed while the rest of the data continue to be restored. This becomes especially attractive when accessing very large files.

Another efficiency problem with BUMP is rooted in the way the database is updated. When the database needs to be updated, a new process is created to perform the update. When only a few update processes are attempting to access the database at the same time, they collide and must wait for each other. A significant amount of time is spent by these processes waiting for the database to become available.

Every time a file is reloaded three processes are created: a reload process, a copy process and a database update process. Process creation in UNIX is relatively fast, but when a large number of files are reloaded at the same time, process creation can consume a significant amount of computer resources and time.

For instance, reloading 15 premigrated files (files that have copies of their data in the online staging area) takes about 20 seconds. Since restoring these files does not require that offline storage be accessed, these files should be reloaded in less than 1 second. All of the time needed to restore these files is spent in process creation and database update collision waits. By eliminating the process creation time and database update collisions, the system's throughput would increase dramatically.

2.3.6 Chosen Limitations

Since almost all of the inactive data in the UNIX filesystem are occupied by regular files, migrating only regular files does not limit the space saving benefits of BUMP.

NFS file servers would benefit enormously from the use of the BUMP system. However, since BUMP provides migration services only to processes running on the local machine, an NFS file server cannot provide the migration system services to its clients. A remote host can access the BUMP system only through UNIX system utilities such as *ftp* and *rcp*, which do not provide the same transparency that NFS provides. Extending the BUMP system to provide NFS support for migrated filesystems would increase its usability.

Providing support to create files larger than the filesystem in which they live would require extensive UNIX kernel and on-disk inode structure modifications. The primary goal of minimizing kernel modifications is probably more important than providing this feature, especially since 2 Gigabyte disk drives, the maximum size of a file in UNIX, are readily available and relatively inexpensive.

2.3.7 Miscellaneous

The UNIX kernel modifications are well-defined, concise, and few. They are nonobtrusive and easy to install.

The algorithm that determines when a file can be migrated to offline storage is very limited; its only inputs are the age and size of the file. If enhanced, a system administrator could better manage online storage. It should include the number of days since last accessed, a minimum size, a maximum size, current size, location, file access patterns and how much data the user already has on offline storage.

The BUMP system suffers from an acute lack of system administrator tools. Tools should be created that facilitate the movement of data through the storage hierarchy, analyze online data for planning purposes, and generate reports on migration system use.

If a user knows that a file will not be accessed in the near future, he may wish to force the file to migrate to offline storage. BUMP does not provide a way to do this. Similarly if a user knows that a file that has been migrated to offline storage will be accessed soon, he or she may wish to reload it before it is needed. The BUMP system does not provide a way for users to "batch" reload files.

BUMP's UNIX kernel modifications do not support the standard UNIX "quota" system. If a site needs to use the BUMP system because it is low on disk space, it is probably already using the quota system. Support should be added for the UNIX quota system.

CHAPTER 3

TRISH ENHANCEMENTS AND EXTENSIONS TO BUMP

The BUMP system was developed to test the feasibility of migrating data from the UNIX filesystem to offline storage. As such it was not overly concerned with system performance or usability. However, in order for a file migration system to be accepted by the user community it must provide good performance, be easy to use, and provide essential features.

The TrISH system was created to address these issues. It takes the fundamental principles developed by the BUMP system to new levels of performance, usability, and reliability. This chapter highlights some of the features of TrISH.

3.1 Goals of the TrISH System

The TrISH system was designed with a number of goals in mind. As was discussed in the introduction, the main goal of TrISH is to provide access to offline storage in a usable way. This implies that it must be integrated into the operating system, transparent to the user, reliable, not add excessive overhead to the access time for the offline devices it manages, and be maintainable by the system administrator.

The base features that were to be added to TrISH are listed in section 1.4. As the changes and enhancements for these features were added to the operating system, other necessary and desirable features became apparent. The merit of these features was compared to the cost of implementing them. If the value of the feature outweighed the cost of implementation, the feature was added (with the understanding that this is a master's thesis and needed to end at some point).

Even though keeping modifications to the kernel at a minimum was not a major concern, they did end up being relatively minor, well-defined, and intuitive.

Features that added performance and reliability were chosen before features that provided usability or bells and whistles, the thought being that, if the data are not safe or if accessing it is miserably slow, then the system will not be used. That said, there were a number of usability enhancements made to the system, both for the end user and for the system administrator.

3.2 Performance Enhancements

System performance is the greatest obstacle to placing BUMP into a production environment. As discussed in section 2.3.5, the BUMP system has some serious performance problems. The TrISH system improves upon the performance of the BUMP system in a number of ways.

3.2.1 Restructuring the System

One of the most serious bottlenecks in the BUMP system is process creation. Every time a file is restored three processes are created: a reload process, a copy process, and a database update process. The UNIX operating system is very good at process creation; nevertheless it is still a time consuming activity. TrISH eliminates most of the process creation overhead through the use of server processes. These processes are always available and waiting for requests from the migration daemon. The most important server processes are the TrISH daemon and the reload servers. The TrISH daemon coordinates the activities of the reload servers, which perform the actual work of reloading files from offline storage.

Not only do these server processes eliminate the extra overhead of process creation, but they also provide other, *even greater*, performance improvements. In the BUMP system, since every offline restore is performed by a separate process, the offline device must always be returned to a "known" state when the process ends. In the TrISH system, the reload server stores state information about the device. With this state information the reload server performs device access optimizations, such as leaving recently accessed media in the device with the assumption that it will be used again soon. For offline devices like optical disk, this results in a big performance gain.

The TrISH migration daemon (`trishd`) listens for requests from the operating system. When a file needs to be restored, `trishd` routes the request to an available reload process. Information, such as the currently mounted offline volume, is communicated between `trishd` and the reload processes. Using this knowledge, the request dispatching algorithms intelligently schedule requests to the appropriate reload process. For instance, if a reload process has the desired offline volume currently mounted, it will receive the reload request. Requests are also prioritized, and all high priority reload requests are processed first. This feature allows low priority "batch" reloads without impacting high priority interactive reload requests.

3.2.2 Concurrent Restore and User Access

While the BUMP system is restoring a migrated file to online disk, the requesting process is blocked from executing. It cannot continue until the entire file has been restored to disk. If the file is large, the process will wait a substantial amount of time. In the TrISH system, the file's data are available to the requesting process as soon as they have been read from the offline device and written to the filesystem. This optimization significantly improves the responsiveness of the TrISH system.

Additionally, if the data are read soon after they have been restored, the read request will be satisfied from the filesystem buffer cache. This improves the responsiveness and performance of the system even more because the number of I/O operations are reduced. This benefits not only the requesting process, but also every process on the system.

3.2.3 On-Disk Data for Migrated Files

The TrISH implementation of concurrent restore facilitates another equally important feature. In the TrISH system, a migrated file can retain a variable amount of data at the front of the file in the online filesystem, even after it has been migrated and gone through the free space creation process. Three benefits of this feature are

- *Reuse of offline data.* When a file is restored to online disk, it is left in a migrated state with all of its data *on-disk*. If the file is not modified, the offline data remain valid, and the file does not need to be remigrated (recopied) to offline storage. It is also immediately available for release by the free space creation process.

- *Fewer false restores.* When a user is searching through a directory using the `head` and `file` commands, migrated files with enough on-disk data will not have to be restored from offline storage if they already contain the necessary piece of the file. Experience has shown that only 8 K-bytes of data must be in the file to prevent the entire file from being restored when using the `head` and `file` commands.

- *Improved interactive performance.* The on-disk data feature can also improve the interactive feel of the system as well. For instance, if the system editors were enhanced to allow the user access to the on-disk data immediately, while the rest of the file is being restored in the background, the user may never know that the file was migrated. This type of service could be especially useful for graphical image files, where the first few thousand bytes of the file would contain a very low resolution markup of the image. The user could view the low resolution image and, if so desired, request that the high resolution image be restored from offline storage.

This single feature (the implementation of on-disk data) has proven to be a very important part of the TrISH system. The ability to reuse offline data and the ability to concurrently access a file as it is being restored are the most important benefits of this feature.

3.2.4 Replacing the Database

The BUMP system uses its own *sequential access* database. The enormous amount of time required to search sequentially through a large database slows down the whole system, making it impossible for the migration daemon to respond to reload requests in a timely manner.

To make matters worse, whenever the database needs to be updated, a database update process is created. When an update process attempts to access the database and another process is currently using it, it waits an arbitrary amount of time (up to 8 seconds) and attempts to access the database again. If the database is still busy, the process waits again. This wait and try loop continues until the database is successfully updated. When a large number of processes are trying to update the database at once, more time is spent waiting for the database than updating it.

The database for the TrISH system has been isolated to well-defined routines that can call any existing database, including commercial, high performance databases. The database access routines allow the TrISH migration daemon to directly update the database without the overhead of creating a new process or the arbitrary waiting involved in the BUMP system. Currently, access routines for an indexed database manager called IDBM have been written. Only a small number of access routines would need to be rewritten to support a different database system.

3.2.5 Better Free Space Management

When a file is migrated in the BUMP system, its data are temporarily stored in the staging area. When the filesystem is low on space, all of the data in the staging area are deleted, thus creating free space in the filesystem. Given these constraints a system administrator has the difficult job of balancing migration parameters.

If *too few* files are migrated and stored in the staging area, free space creation can be very slow. The reason is that rather than just releasing the online space occupied by migrated files in the staging area, the entire migration process, from identifying files to copying them to offline storage, must be performed whenever space is needed in the filesystem. If, again, too few files were migrated, the whole process would need to be performed over again.

If *too many* files are migrated and stored in the staging area, then when the staging area space is released, more free space would be created than is needed to fulfill the request at hand. The problem lies in the fact that all of the files whose staging area data were deleted will need to be restored from offline storage next time they are accessed.

Moreover, BUMP selects files for migration based solely on the value of the file's eligibility and not on the space needs of the filesystem. If the migration process finds an old file in the filesystem, it will be migrated regardless of the current amount of free space. Unless the data stored in the filesystem are very regular and predictable, tuning the migration eligibility algorithm is very difficult.

To address these problems, TrISH includes three additional system administrator defined parameters: the high watermark, the low watermark, and the migrated target. These values are used as follows:

The high watermark When the amount of used space in the filesystem reaches the high watermark, the free space creation process is started.

The low watermark The free space creation process releases space in filesystem until the amount used space in the filesystem is at or below the low watermark.

The releasable watermark The nightly migration process attempts to migrate a sufficient number of files so that the amount of used space minus the amount of "releasable" space in the filesystem is at most this value. (Given the migration eligibility algorithm, this may or may not be possible.)

These values are expressed as a percentage of the total space in the filesystem. These three values can be tuned by the system administrator to meet the needs of the site. If a filesystem's free-space is off, either too low or too high, the high and low watermark values can be adjusted so that the free-space creation process creates just the right amount of free space. If a system generally converts a large amount of migrated space into free space, then the releasable watermark value should be lowered. If a system rarely converts migrated data to free space, this value can be increased.

3.3 Usability Enhancements

A system that solves only half of a site's storage needs is only marginally better than no system at all. The TrISH system builds upon the solutions provided by the BUMP system to provide a more functional, extendable, and integrated storage management system.

3.3.1 NFS Support

As workstations continue to become more powerful and less expensive, more and more sites are moving away from a central computer environment to a distributed computing environment. Increasing network speeds and availability accelerates and encourages this trend. These distributed environments commonly use centralized file servers which are accessed through the use of network file access protocols such as NFS to store their data. These file servers would benefit greatly from the services a file migration system can provide. The TrISH system supports access to migrated files through NFS.

3.3.2 Offline Device Support

One of the goals of the old BUMP project was to facilitate easily the addition of new offline device types into the storage hierarchy. BUMP was only marginally successful in reaching this goal. Online and near-line devices do not fit into the BUMP device abstraction which was designed with the assumption that only magnetic tape devices would be used. Many of the device specific functions are performed by the BUMP system utilities rather than the device access method routines. For instance, CRC checking and granule processing is performed by all BUMP system utilities. Because of this, if functions other than CRC checking and granule processing would be more appropriate for a particular device, they cannot be provided. Additionally, because device state information is kept by the BUMP applications rather than by the device access routines, some state information required by robotic auto-loaders is not available. For these reasons it is difficult to add support for near-line devices to the BUMP system.

The TrISH system uses a higher level of abstraction for offline storage devices. It is the responsibility of the device support routines to perform CRC checks, granule allocations, and other data processing that is appropriate for the device. The TrISH utility programs use these high-level device access routines and, hence, can use new storage devices without modification. This higher level of abstraction also enables the device support routines to perform intelligent media management. For instance, keeping the last used tape in the drive until it is known that it is no longer required.

Initially, device access routines have been written for an "online compression" device, an "online copy" device, and the Hewlett-Packard rewritable magneto-optical disk autochanger. The optical disk device access routines can optionally perform software data compression as the data are written to the device.

3.3.3 Site Integration Support

The TrISH migration system requires a number of services that may or may not be presently available at a particular site. For instance, TrISH requires the services of a database system to store information about migrated files. If a site has an existing database system they are familiar with, then TrISH should be able to use it. If, on the other hand, the site does not have one, one should be provided by the TrISH system. The ability to integrate existing services into the TrISH system improves the probability that it will be accepted by the support staff.

TrISH has separated the required services from itself so that a site may easily "plug in" their own services to replace the TrISH provided ones. A well-defined and separate interface has been provided so that integration with existing systems will be painless.

The replaceable services that TrISH requires are as follows:

- Modified kernel. A small set of kernel support routines provides the interface between the migration daemon (`trishd`) and the operating system. Any operating system kernel that supports these routines will be able to use the TrISH migration system.

- Database server. The database provides the information storage and retrieval functionality necessary to keep track of migrated files and their offline data.

TrISH comes with support for the IDBM database, but support for other databases would be easy to add.

- Media manager. The media manager provides a way for TrISH to request another unit of offline media (such as another blank tape). TrISH has a simple media allocation system. Access to another system would be easy to add.

- Device manager. The device manager controls access to offline devices. If TrISH needs to read a tape, it must first acquire the exclusive use of a tape drive. The device manager provides this service by serializing access to offline devices. The device manager also provides the support necessary for TrISH to use nondedicated offline devices. The device manager only allocates and assigns devices to processes; it does not perform the actual I/O to the device. The device manager service eliminates the restriction in BUMP that all devices must be dedicated to the BUMP system and cannot be used by other processes.

- Message manager. When a message needs to be sent to the operator or to the system administrator, the message manager is used. The message manager may log the message or distribute copies of the message to various people. An X-windows-based message system has been provided with TrISH, but integrating a new system would be easy to do.

3.3.4 Eligibility Algorithm

To give the system administrator control over which files are chosen for migration, the eligibility algorithm was written as a C language routine that can be easily modified to meet the needs of the site. The eligibility algorithm is used to decide not only which files to migrate but also which files to release during the free space creation process. The eligibility algorithm has access to the following data:

- Name of the file

- The `fstat` data for the file, including userid, user group, size of file and last access times

- The TrISH file system configuration parameters. Additionally these parameters can be extended to include custom values defined by the system administrator
- The amount of on-disk data for files that have already been migrated (used by the free space creation process)

With access to these data, the eligibility algorithm can be modified to implement any number of site-specified migration policies. Some potential site policies are as follows:

- Larger files should be migrated before smaller files.
- If a file is smaller than some minimum size, do not bother to migrate it.
- Active files stay; inactive files are migrated.
- Files that are infrequently modified, even though they may be frequently read, are migrated whereas files that are frequently modified are not.
- A particular user or group gets preferential treatment. Their files will be larger and older than another group's files before they are migrated.
- Certain types of files should be migrated before other types of files. For example, object files should be migrated before source files.
- Files with a given name or within a given directory tree should be migrated before others. Files in another directory tree should never be migrated.

The eligibility algorithm assigns a "badness value" to each file in the filesystem. The files are then ranked and sorted by their badness values, with the largest badness value ranking first and the smallest badness value ranking last. The ordering of files by their badness value is used during both the the migration process and the free space creation processes. These processes are be described in section 4.1.

3.3.5 System Administrator Tools

TrISH, unlike the BUMP system, has a number of tools designed to be used by the system administrator. With these tools the administrator can analyze the needs of a filesystem before enabling file migration, gather statistics on the effectiveness of the migration system, monitor the work load of the system, and forecast future needs.

A filesystem analysis and graphing tool will help an administrator choose appropriate migration system parameters. This tool gathers information about the number, age, and size of files in the filesystem. This tool would be used to define initial values for migration system parameters and to verify and tune those parameters on an ongoing basis.

A tool that examines migration system statistics within the UNIX kernel and shows access patterns to migrated and nonmigrated files will help an administrator maintain good migration system parameters. These kernel statistics include calls to the `open`, `read`, `write`, `trunc`, `unlink`, and `getattr` operating system calls for both migrated and nonmigrated files. For migrated files, the amount of time spent waiting for file reloads is also tracked. This information can be used to analyze the effectiveness of TrISH.

The `trishd` process logs all requests. A log processing program enables the system administrator to analyze the type and volume of migration system requests. These data will give the administrator even more detailed information on how the migration system is being used.

3.3.6 User Tools

The users of the system will often know when they will or will not be using a particular file in the near future. TrISH provides a few tools that enable the user to control and direct the handling of files. The following tools and commands were not provided by the BUMP system.

A user has the ability to force the migration of files to offline storage using the `trforce` command. This is very useful if a large file has just been created and it is known that the file will not be needed in the near future. By forcing such a file to migrate, the user will create free space in the filesystem and will reduce the chance of the user's other files, which may be needed soon, from being migrated.

The user also has the ability to release on-disk space being held by migrated files. Remember that migrated files can have data in them waiting to be released when needed. Using the `trelease` command, the user can release the space in migrated files. Using this command, the user can free up space in the filesystem and possibly keep another file from having its data released.

Using the `trctl` command, a user can perform batch reloads of migrated files. This is very useful if it is known that the user will be needing a set of files and wants them to be reloaded in advance. The batched reload requests are placed on a low priority queue and are processed in the background when there are no other higher priority requests.

TrISH also implements the migration system process flags discussed in Chapter 2. Using these flags, the user can tailor the way certain conditions are handled. For instance, the user can tell the system to return an error, rather than automatically reloading migrated files, when the process attempts to read past the on-disk data. The user can also request that his or her process never receive an *out-of-space* error message but rather blocks until the migration system has a chance to free some disk space for it to use.

3.3.7 Miscellaneous

The UNIX Quota system is used by system administrators to help manage online disk usage. With it a system administrator can limit the amount of disk space each user may consume. Considering that a site which is currently short on storage space is in all likelihood using the quota system and considering that these are the very sites that would benefit most from implementing the TrISH system, support has been added for the quota system. Data residing in the filesystem are deducted from the user's quota. When a migrated file's data are released, that space is no longer counted against their quota.

The TrISH system has one central configuration file where system parameters are stored and can be easily maintained by the system administrator. This contrasts with the BUMP system in which many configuration parameters are hard-coded into the source code. An additional feature is that the configuration file can be extended. If a site adds a new feature to the TrISH system, the configuration file can be extended to contain the site specific values used by the new extentions to TrISH. The TrISH configuration file is described it detail in Appendix A.

Key algorithms, like the badness calculation function used to order files for migration, have been separated from the rest of the source code to enable easy customization of the system. These algorithms can be quickly located and easily modified to meet site specific needs. Combined with extensions to the configuration file, this enables the system to be highly tuned and customized for a site's special needs.

The operating system calls `ioctl` and `select` are used by programs to gather information about the status of open files. These system calls have been modified to understand and return information about migrated files. The `ioctl` system call returns the amount of on-disk data available to be read from the file. The `select` system call returns true when an open migrated file has data available to read.

3.4 Reliability Enhancements

The data reliability of the BUMP system is quite good. Much thought and effort were put into constructing a system that would protect and keep safe the user's data. It runs CRC checks on all data written to and read from offline storage. It provides effective recovery techniques to deal with media errors. Its databases were built so that even if the BUMP system was totally destroyed, the system administrator could, by hand, rebuild the system and recover all of the data.

TrISH has retained all of the reliability features of BUMP (CRC checking, granule processing, multiple offline copies, error recovery) and has added one more reliability enhancement.

The only problem with the BUMP system's reliability is the centralized database. If the database is corrupted or lost, the entire BUMP system is useless. In the TrISH system there are separate databases for each of the managed filesystems, rather than one monolithic database. The databases for each filesystem hold only the information for the files migrated from that filesystem. When one filesystem's database is unavailable, no other filesystems are affected. An added benefit to this architecture is that a filesystem is self-contained. It, along with its database, can be moved to another machine with very little work. If a filesystem's databases are located within itself, the backup and restore process is also simplified. This also prevents the database from being out of date with the filesystem, should a filesystem need to be restored from its backups.

3.5 Summary of Enhancements to BUMP

A number of enhancements over the BUMP system have been added to TrISH. A summary of the major enhancements in the TrISH system are listed below.

- Eliminated excessive and unnecessary process creation.
- Created intelligent reload servers.
- Created an intelligent migration daemon (`trishd`).
- Enhanced device access routines and methods, enabling smarter handling of near-line devices.
- Implemented concurrent restore and user access.
- Facilitated the reuse of offline data.
- Provided support for on-disk data.
- Improved interactive performance.
- Isolated database access routines from the TrISH system.
- Implemented better free space management through the use of high, low and releasable watermarks.
- Provided support for NFS server functions.
- Implemented a media manager.

- Provided a device manager.
- Furnished an operator messaging manager.
- Enhanced badness value calculation function.
- Implemented filesystem analysis tools to determine if file migration is appropriate for a particular filesystem.
- Provided kernel statistics gathering tools.
- Enabled migration log analysis facilities.
- Enhanced the migration daemon
- Provided the reloader control program.
- Enabled a file to be forced to migrate.
- Forced release of migrated file.
- Implemented batch reloading of migrated and released files.
- Facilitated support for the standard quota system.
- Designed the central extensible configuration file.
- Enhanced system to work with distributed database system.
- Extended system calls to understand migrated files.

CHAPTER 4

TRISH IMPLEMENTATION DETAILS

The goals of the TrISH system were discussed in detail in Chapters 1 and 3. To summarize, they are to provide easier access to offline storage through the method of automatically and transparently moving large inactive files from expensive online storage to inexpensive offline storage and to provide this functionality in an efficient, reliable, integrated, and feature-rich way.

4.1 TrISH Operation

To accomplish the goals set out for it, the TrISH system transparently migrates a file's data to offline storage when they are not being used and transparently restores the data when they are again being used. The steps involved in migrating and restoring a file's data are described below.

1. Identify a file that should be migrated.
2. Make the file into a migrated file.
3. Copy the file's data to offline storage.
4. Mark the file as being releasable.
5. Release the file's online storage when space is needed.
6. Restore the file's data from offline storage when they are accessed.
7. De-migrate the file when it is modified.

To identify files that can and should be migrated, the target filesystem is scanned and its files are assigned a "badness value" using a system administrator defined badness function (explained in detail in section 3.3.4). The files are then ordered by their badness values and the files at the top of the list are migrated. Files are migrated until the *releasable watermark* value has been achieved or until no more files meet the minimum requirements. The initial state for newly migrated files is *nonreleasable* with all *on-disk* data. Because the file's data have not yet been copied to offline storage, its data cannot be released for use by another file. Furthermore since none of the file's data have been released, they all reside on-disk.

At this point, the file's data are copied to offline storage. The system administrator, through configuration file parameters, specifies which offline devices the data are copied to and the number of copies to be made. After the file's data are successfully copied to offline storage, they are marked *releasable*, meaning its data can be released by the free-space creation process and made available to other files.

When the used space in the filesystem rises above the *high watermark*, the free space creation process is started. The releasable files in the filesystem are ordered by their badness values. Files at the top of the list are selected and their on-disk data are released until the *low watermark* value is reached or until there are no more releasable files. If, during the migration and copy-out processes above, a sufficient number of files have been migrated and copied to offline storage, the free space creation process can occur a number of times before the migration process (steps 1-4 above) must be dispatched again. The idea is to migrate a large amount of space during off peak hours so that it can be easily released and used during peak hours when it is needed.

As alluded to above, when a migrated file is accessed, the offline data must be restored to the online filesystem. The responsibility of restoring the file belongs to the TrISH migration daemon and its reload processes. When a file needs to be restored, the kernel notifies the migration daemon, which in turn dispatches the reload request to an available reload process.

If a migrated file is modified, it is changed from a migrated file back to a regular file. The old offline copies are no longer valid and can be discarded. To become a migrated file again, it must once again go through the migration process. If, however, the file is not modified, it retains its status as a migrated and releasable file. Its disk space can be released at any time since the existing offline copies are still valid.

These processes are described in greater detail in the appendixes. The identify, migrate, and copy-out processes are described in Appendix B. The free space creation process is described in Appendix C. The restore process is described in Appendix D.

4.2 UNIX Kernel Support

A number of enhancements were made to the UNIX operating system kernel to support the process of file migration. The modifications are independent of the migration system, allowing any migration daemon to use the same operating system interface. The changes and enhancements to the UNIX kernel can be categorized as follows:

- Enhanced the filesystem to support migrated files.
- Created new system calls and enhanced existing system calls to create, manage, and monitor migrated files. The new and enhanced system calls are discussed in detail in Appendix E.
- Provided a communication path between the UNIX kernel and the migration daemon.
- Enabled special migration system process flags to control the level of transparency.
- Provided support for NFS access.
- Added support for the quota system.

The changes are described in greater detail in the following sections.

4.2.1 Filesystem Support

To support migrated files, the standard UNIX filesystem data structures were enhanced. A field was added to the filesystem super block that contains the filesystem's *high watermark*. When the amount of used space in the filesystem reaches this point, the free space creation process is started.

A new file type, `IFMIG`, was created and the on-disk inode structure was enhanced to include two additional fields. The new fields are `i_fmigid` and `i_ondisk`. The `i_fmigid` field holds the file's migration identifier. This identifier is used to track the file's data through offline storage and to identify it in the migration system databases. The `i_ondisk` field contains the amount of on-disk space in the file. Additionally, the previously unused on-disk inode field, `i_flags`, is now used to hold the `IFMIGREL` and `IFMIGMOD` flags. They indicate that the file is releasable or contains modified data, respectively.

In addition to changes to the filesystem data structures, changes were made to the filesystem modules in the operating system. When a migrated file is opened, the operating system notifies the migration daemon by sending it the file migration identifier, the "`fmigid`," of the file. The migration daemon can choose to begin reloading the file immediately, or it can postpone reloading the file until a read request is made for data that do not reside on-disk.

When a process tries to access data that do not reside on-disk, the process is blocked and a message indicating that the process is waiting for the file to be reloaded is sent to the migration daemon. The migration daemon is responsible for reloading the file as soon as possible, so that the blocked process can resume execution. As the file's data are restored, the blocked process is allowed to continue so that the recently restored data can be processed. If more than one process is reading the file at a time, only one reload request is sent to the migration daemon, and all processes are unblocked when data are available.

When a migrated file is closed, the migration daemon is notified. It has the option of canceling any pending or in-progress reload request for the file. This could happen if the file was opened but never read or if only the first part of the file was read.

When a write request is made to a migrated file, the write is allowed to complete when the on-disk data include the part of the file that is being written. The `IFMIGMOD` flag, in the `i_flags` field is set, indicating that the file's data have changed and that it should be de-migrated (made into a regular file) when it has been completely restored. A file that is truncated to a nonzero length is handled the same way. When a file is truncated to zero length, it is immediately made into a regular, zero length file.

When a file is modified, truncated, or deleted, the offline copies of the data are no longer needed. A message is sent to the migration daemon notifying it that it can discard the offline data and invalidate or delete the file's database entries.

If a migrated file containing an executable program is to be run, the process is blocked until the entire file, or program, has be restored to on-disk storage. It is necessary to wait until the entire program is reloaded because the UNIX kernel will demand page the program directly out of the filesystem and into main memory[9], bypassing the standard filesystem code.

The filesystem's space allocation routines were also modified. These changes effect not only migrated files but regular files as well. When an allocation request causes the used space in the filesystem to rise above the high watermark, the migration daemon is notified so that it can start the free space creation process. Additionally, when an allocation request would normally return with an `ENOSPACE` error message because the free space in the filesystem has been exhausted, the requesting process is instead blocked. An urgent out-of-space message is sent to the migration daemon. The process remains blocked until enough space has been created to honor the allocation request. If the free space creation process was unable to create some free space, the allocation request fails with an `ENOSPACE` error message.

4.2.2 Controlling Transparency

By default, migrated files appear to be regular files. Except for possible access delays, unmodified programs cannot tell when they are accessing a migrated file.

A process can customize the default behavior of the system when accessing migrated files by setting new process flags. These process flags are passed from parent processes to their child processes, and, of course, the child process can modify them as well. The process flags are as follows:

FMIG_FLAG_NOTRANSP By setting this flag, a process will be notified with an error message whenever a read or write request to a migrated file would have blocked. Additionally, a background reload request for the file is sent to the migration daemon.

FMIG_FLAG_CANCEL If this flag is set, the background reload request normally generated when the `FMIG_FLAG_NOTRANSP` flag is set is suppressed. This flag only has meaning when used in conjunction with the `FMIG_FLAG_NOTRANSP` flag.

FMIG_FLAG_SPACERETRY By setting this flag, the process will be blocked rather than receive an ENOSPACE error when the filesystem does not have enough free space to satisfy an allocation request. The process will remain blocked until enough free space has been created in the filesystem to satisfy the request.

4.2.3 Migration Daemon Communication Path

A communication path, implemented as a pseudo device driver in the kernel, is used by the file migration routines in the UNIX kernel and the migration daemon to send and receive messages. The regular UNIX system calls, `open()`, `read()`, `write()`, and `select()`, are used by the migration daemon to open communications, receive messages, send messages, and test for messages from the kernel.

When the migration daemon is running and operational, the communication path is open. The kernel assumes that when the path is open a migration daemon is waiting to process requests, and when it is not open, no migration daemon is running. With this assumption, file migration activities are enabled while the communication path is open. When it is not open, some of the file migration activities are disabled. For instance, if an attempt is made to delete a migrated file while no migration daemon is running, the request is denied, since the migration daemon needs to be notified when a migrated file is deleted. On the other hand, a migrated file's on-disk space can be released when no daemon is running because the migration daemon does not need to be notified. An `errno` value of `EFMIGOFF` is returned when a request is canceled or aborted because the migration communication path is closed.

The structure of the messages sent across this communication path is discussed in greater detail in Appendix F. Included in this discussion is a list of the valid operations.

4.2.4 NFS Server Support

Because of the structure of the TrISH operating system modifications, no change to the NFS server code was required to enable NFS access to migrated files. When a request is made from NFS to the UNIX filesystem, the standard filesystem routines are used. Since these standard routines were already enhanced to handle migrated files, the NFS server had immediate access to migrated files.

A few restrictions, however, exist on the method and the level of access to migrated files through NFS. NFS clients must use *hard*, rather than *soft*, mounts when mounting TrISH enabled filesystems via NFS. This restriction is required because the length of time needed to reload a migrated file is often greater than the time-out limit for soft-mounted filesystems. Hard-mounted NFS filesystems, on the other hand, do not time-out NFS requests and hence can deal with long delays.

Because process flags are not passed between the NFS client and the NFS server, the transparency controls discussed in section 4.2.2 are not supported on NFS mounted filesystems.

The NFS protocol has specified the information that is sent between the client and the server when, for instance, performing a `stat` system call. The specification precludes appending additional information for migrated files. As a result, migrated files always look like regular files to NFS clients, and they are unable to use the `select()` and `ioctl()` system calls discussed in Appendix E.

A hidden benefit, however, is lurking in these restrictions. Because the NFS client machines do not need and, in fact, cannot get any information about migrated files they are guaranteed transparent access to them. This allows an NFS client, whose operating system does not contain the TrISH enhancements, access to the migrated files on a TrISH-enabled NFS server.

In a future project, enhanced access to migrated files via NFS could be developed. At this time, however, it is beyond the scope of this project to modify the NFS protocol to provide these enhancements. Transparent access to migrated files is sufficient for this project and is, in fact, a giant step beyond what the BUMP system provided.

4.2.5 Quota System Support

Like the NFS server support discussed above, support for the quota system required no kernel changes. This is because the TrISH kernel modifications make use of the standard filesystem routines, which already contain support for the quota system.

4.3 Device Access Methods

One of the goals of the TrISH system is to allow the easy integration of new offline devices into the storage hierarchy. To this end a small set of well-defined access routines has been implemented. Integration of a new device into the TrISH storage hierarchy simply requires definition of the access routines and an update to the configuration file. No programs need to be changed. The migration and reload processes will access the new devices through their access methods.

Another goal of the TrISH system is to reliably and efficiently manage offline storage. To aid in achieving this goal, offline data are organized into "granules." A file may be broken down into multiple granules, and one given granules may be written to one or more offline storage devices. Granule size is determined by the access method routines and is defined to optimize both the use of the media and the performance of the device.

Using granules increases the reliability of the system by providing an easy mechanism of recovering from media and device failure. If an offline granule is unusable, another granule containing the same piece of data can be used instead. Granules also make the task of writing access methods easier. Because the size of a single chunk of offline data is reduced, the access method is liberated from the difficult task of handling multivolume datasets. The granule implementation has already provided the mechanism for splitting a file's data into manageable pieces.

The device access routines can be grouped into the following categories: configuration functions, initialization and cleanup functions, volume handling functions, granule handling functions, and data block functions. Some functions, if they are not needed or are not applicable for the device, do not need to be defined. For instance, the *compress* method does not mount or unmount any media and does not have those functions defined. The TrISH device access routines are discussed in further detail in Appendix G.

To date, access method routines for copying data to another filesystem automatically compressing data inside the filesystem, and copying data to the Hewlett-Packard magneto optical jukebox has been developed. Providing access routines for either robot or operator mounted magnetic tape devices would be a straight-forward addition to the system.

4.4 Database Access Functions

Database system independence is another one of the goals of the TrISH system. Because there are many types of database systems, including relational databases, indexed files, and sequential files, access to the database has been abstracted into functional units of work. For instance, when the reload process needs a list of granules containing a file's data, it calls the `granuals_for_fmigid()` function. Using the database system on the host, this routine creates a list of the granules containing the required data.

There are database access functions for initializing and closing the database, adding, deleting, updating, querying, and sorting various database entries. The types of database entries include granule entries, volume entries, releasable file entries, migratable file entries, and forced migration file entries. A list of the database access functions can be found in Appendix H.

Database system reliability is another design goal of the TrISH system. This is accomplished, in part, by separating the database entries for each migrated file system. If the database for one file system is damaged, the other file systems on the machine can still be used while the damaged database is reconstructed. The database access functions have been designed and implemented with the expectation that each filesystem has its own database.

A side benefit of this technique is that, depending on the implementation of the database, the performance of the overall system increases. Since each filesystem has its own database, it is smaller, easier to maintain, and easier to optimize. Additionally, since there are multiple databases on the system, the database accesses have inherent parallelism.

A filesystem's database is divided into two different logical databases, the main TrISH database and the "out-migration" database. The main TrISH database is where permanent information is stored, such as the granule and volume database entries. The out-migration database holds the list of files to migrate along with their badness values. These databases are separated because the out-migration database is deleted and re-created on a regular basis to reflect updated badness values as well as new files in the filesystem.

4.5 The TrISH Migration Daemon

The TrISH migration daemon, `trishd`, orchestrates the activities of a number of different entities. It listens for requests from the kernel, the system administrator, and the users. It dictates the activities of the reload and free space creation processes. It sets parameters in the filesystems and enables the file migration activities of the UNIX kernel. In short it is the heart and brains of the migration system.

4.5.1 Request Management

The TrISH daemon keeps track of requests on three different priority queues. The highest priority queue is where kernel requests are placed, the lowest priority queue is for general user requests, and the medium queue is for high priority user requests. Lower priority requests are dispatched only if no higher priority request is dispatchable. Once a request is dispatched it will not be interrupted or preempted, even if a higher priority request is dispatchable.

In the TrISH system, each offline device has its own reload server, and each reload server services only one device. In order for TrISH to start using a device, the corresponding reload server must be started. The reload server is shutdown to stop using a device. To `trishd`, there is no distinction between a device and reload server.

The file migration routines in the UNIX kernel send reload and out-of-space requests to `trishd`. Reload requests are placed on the high priority work queue and dispatched when a reload server is available. If a reload server is not able to successfully reload a file, the reload request is requeued, and a new reload server is given the chance to restore the file. By doing this, all possible ways of reloading a file are explored before returning a *reload failed* message to the kernel.

Out-of-space requests are immediately handled by starting a free space creation process. When the process sends back a message indicating that some space was created in the filesystem, `trishd` sends a message to the kernel informing it that free space was created and that any blocked processes should be restarted. If the freer process is not successful at creating space in the filesystem, a failure message is sent to the kernel, and the kernel in turn returns error messages to processes waiting for space to be created. Because a single filesystem can potentially generate a number of out-of-space requests in a short amount of time, `trishd` must keep track of current free space processes, and only create a new process if one does not already exist for the filesystem. The free space creation process is discussed in further detail in Appendix C.

Every request that is sent to `trishd` is forwarded to the TrISH log filter program.

The current filter program just logs all of the requests to a file, but future intelligent log filters could analyze the requests and watch for patterns of access. For instance, if more than a few files were accessed in the same directory, the log filter could generate reload requests for all of the migrated files in the directory under the assumption that the user will probably request the files in the near future.

The system administrator manages `trishd` by querying the state of reload servers, starting and stopping reload servers, querying the progress of reload requests, cancelling reload requests, and shutting down the TrISH system. When a device is started, its controlling reload process is created. The reload process is responsible for sending regular status reports to `trishd` so the current state of the reload process and the device is known. This information is displayed when a query is received concerning the state of a reload process. When a request is made to shutdown a reload server, `trishd` removes the reloader from the available queue and sends it a shutdown message. The reloader is then responsible for ending gracefully.

If a reload request is currently dispatched when a *cancel reload* request is received, the request is forwarded on to the reload server. When the server reports back to `trishd` that the cancel request was completed, the original reload request is deleted from the queue. If the reload request was not dispatched, then the original reload request is just deleted.

A user can query the state of reload requests, cancel reload requests, and initiate batch reload requests for files. These requests are handled the same way that system administrator requests are handled, except a user can only cancel reload requests for his or her own files and can only initiate reload requests for files he or she has access to.

The system administrator and the system users communicate with `trishd` with the `trctl` program. It is how the system administrator starts, stops, and retrieves information about reload servers and shuts down the TrISH system. It is also how a user initiates batch reloads, cancels pending and in-process reload requests, and checks on the status of reload requests. The `trctl` program is discussed in further detail in Appendix J.

4.5.2 TrISH Reload Servers

The TrISH reload servers are responsible for listening for, and responding to, control messages from `trishd`. These messages include reload requests, cancel reload requests, status update requests, shutdown requests, and regular "pings" to make sure that the reload server is still alive. The reload server must be able to accept requests at any time, even when it is busy responding to a previous request. For instance, a *cancel* request could arrive while the reloader is busy restoring a file. The list of the valid reloader requests is in Appendix D.

There are a number of tasks performed by `trishd` in the management of reload servers beside just starting and stopping them. For instance, reload servers are responsible for sending periodic status information. Included in this information is the current volume mounted on the device, the current file being reloaded including its `fmigid`, the owner and group of the file, the total size of the file, and the amount of data that have been restored to the file.

The name of the currently mounted volume is used by `trishd` to perform dispatching optimizations. Since most offline devices, including robot controlled devices, typically suffer from long delays when mounting new volumes, `trishd` groups reload requests for the same offline volume to the same device. This enables full utilization of a volume once it is mounted. As mentioned above, high priority requests are serviced before low priority requests, even when the low priority requests are on the same volume as one of the high priority requests.

Occasionally, a reload server will be unable to reload the requested file. When this happens, `trishd` will automatically requeue the request, and send it to another reload server that may have access to other offline copies of the file. A pathological case for optical disk juke-boxes that `trishd` must deal with is the situation where two CD drives are available for use, one drive has volume 1A mounted and the other drive attempts to mount volume 1B. Unfortunately, volume 1B is on the flip-side of volume 1A. So even though volume 1B is not mounted, this request can only be serviced when volume 1A is dismounted. `Trishd` handles this situation by requeuing the reload request when the reload process returns a "volume busy" error message and only dispatching when the conflicting volume is unmounted.

A feature called "nondedicated devices" enables TrISH only to use a device when it is needed. The nondedicated reload server is not started when `trishd` initializes but rather remains in a *dormant* state until a reload request is to be dispatched to it. The reload server is started and given the reload request. When no more reload requests exist for the device and it has been idle for a short period of time (configurable by the system administrator), the reload server is again shutdown and placed in the dormant state waiting for another reload request. While the server is in the dormant state, the device is unallocated and available for use by other, non-TrISH, processes on the computer system. This feature is especially useful when more than one device of a particular type exists and some of these devices should, for most of the time, be available for non-TrISH system work, but should also be used by TrISH when a large number of reload requests are received.

4.6 Enhancements to Standard Utilities

A few of the standard UNIX utilities that access and provide information about files and filesystems have been enhanced to return information about migrated files. The enhanced utilities are discussed in this section.

4.6.1 The `ls` Command

There are situations where users will need to know information about migrated files. For instance, a user may want to know how much of a file's data resides in on-disk space. The `ls` command has been enhanced to return the `fmigid` value, and the amount of on-disk space when the "`-l`" flag is specified.

The example below shows three migrated files. The first file's data all resides in on-disk space, the second has 1K of on-disk data, and the last has none. The type of these files is listed as "m," meaning migrated, and their `fmigid` is listed along with the on-disk space.

```
65 saaz> ls -l big* test*
mrw-rw-r-- 1 bytheway 18186872 Jan 30 18:59 bigfile1 [102-13582 18186872]
mrw-rw-r-- 1 bytheway 34615296 Jan 30 20:49 bigfile2 [102-13590     1024]
```

```
mrw-rw-r-- 1 bytheway  4111128 Jan 28 01:57 testing  [102-13581        0]
```

4.6.2 The `find` Command

A new `-type m` flag has been added to the find command to locate migrated files. The `-type f` flag only finds regular, nonmigrated files. When the `-ls` flag is set, the output lists a file type of "`m`" for migrated files. The `-ls` flag to find, unlike the `ls` command, does not list the file's `fmigid` or the amount of on-disk space.

4.6.3 The `fsck` Command

The `fsck` command was modified to ignore the high watermark value in the filesystem super block. It was enhanced to know that the migrated file type (`IFMIG`) is a valid type and that migrated files should be treated the same as regular files. Since migrated files use the data block pointers in the inode the same way that regular files do, only these simple changes were needed for the `fsck` program.

4.6.4 The `df` Command

A new flag "`-m`" was added to the `df` command. When this flag is specified, the `df` command queries the database of the filesystem to get the amount of releasable space in the filesystem. The amount of releasable space is added to the amount of available space. This value is then subtracted from the amount of used space. By specifying the `-m` flag, a user can see the true amount of available space in the file system. Below is an example of `df` with and without the `-m` flag.

```
86 saaz> df /u
Filesystem           kbytes    used   avail capacity  Mounted on
/dev/sd0f            193870  126731   47752    73%    /u

87 saaz> df -m /u
Filesystem           kbytes    used   avail capacity  Mounted on
/dev/sd0f            193870   23778  150705    14%    /u
```

4.6.5 The `du` command

A new flag "`-m`" was added to the `du` command. When this flag is specified, another column is printed containing the amount of space in the file/directory that is *not* releasable. By using this flag the user can find out how much nonreleasable space is being used by files. Below is an example of `du` with and without the `-m` flag.

```
99 saaz> du -s *
17788   bigfile1
1       bigfile2
23      trishlog
2028    trishlog-11-11-93
1688    trishlog-11-25
2408    trishlog-11-9-93
4020    trishlog-9-17-93

100 saaz> du -s -m *
17788   0       bigfile1
1       0       bigfile2
23      23      trishlog
2028    0       trishlog-11-11-93
1688    0       trishlog-11-25
2408    0       trishlog-11-9-93
4020    0       trishlog-9-17-93
```

4.6.6 The `dump` Command

The `dump` command was enhanced so that it would treat migrated files the same way it treats regular files. No other changes were necessary to support migrated files.

4.6.7 The `restore` Command

The `restore` command was enhanced so that it will correctly restore migrated files. To restore correctly a migrated file, the `fmigid` field, the releasable flag (`IFMIGREL`), and the modified flag (`IFMIGMOD`) must be set correctly. Additionally, if part of the file was not on-disk when the dump tape was created, that part of the file must be released since the dump tape does not contain valid data for that part of the file.

The steps to restore a migrated files are as follows:

1. If the on-disk space was less than the size of the file, then extend the file to the correct length by seeking to the end of the file and writing a dummy byte

of data (these data will be released later). If this is not done, the file's length would be incorrectly set to the length of only the on-disk data.

2. Use the `fmig_migrate()` system call to make the file into a migrated file. Use the same `fmigid` that was used in the old file. With the `fmigid` in place, the offline data are again associated with the file.

3. If the releasable flag (`IFMIGREL`) was turned on in the old file, set it on in the newly restored file using the `fmig_releasable()` system call.

4. If the on-disk space was less than the size of the file, then release the file's on-disk space using the `fmig_release()` system call.

5. If the modified data flag (`IFMIGMOD`) was set, cause the kernel to reset this flag by reading and rewriting the first byte of the file.

CHAPTER 5

ANALYSIS OF TRISH

Many of the problems (described in Chapter 2) with the BUMP system are a result of design decisions that limited the capabilities of the system. The TrISH system was redesigned in a number of fundamental areas to address these problems. As a result of the improved design, the TrISH system does not have many of the limitations that the BUMP system suffered. Some of the original limitations went away with no extra work. Interestingly, requirements for some of the solved problems, like the quota system support, were not fed into the design phase. This gives validity to the design of the TrISH system and gives some assurance that it is well-integrated with and conforms to the fundamental design of the UNIX operating system.

5.1 TrISH Design

As was mentioned in Chapter 3, the design decision to allow a file to have both on-disk data, and offline data, had a number of system performance benefits. Interestingly enough, the performance gains were not in the area first thought to be most important. These performance gains will be discussed in section 5.2.

Redesigning the offline device access method routines was a formidable but successful undertaking. The task of identifying and abstracting the minimum basic functions was difficult. However, by using the TrISH access method routines, programs that access data on offline media are greatly simplified, and more importantly, they are more generic. They no longer have to worry about mounting offline media, recovering from media error, generating CRC check sums, and handling device specific routines; the access method routines perform these tasks. Additionally, since the abstraction boils down to easily understood and implemented pieces, writing new device access methods is also simplified. Unlike the BUMP device access routines, which often required applications to do “special case” coding for devices that did not quite fit the abstraction model, the TrISH device access routines require no application level special case coding.

The decision to base the TrISH enabled filesystem on the standard UNIX filesystem and its inodes simplified the design of the system greatly. The underlying filesystem provides the directory structure, the data block allocation routines, the online device access routines, and special functions like enforcement of quotas. On the other hand, this decision also implies some limits on the implementation. For instance, because TrISH is inode-based, the maximum number of files in the filesystem is limited to the number of inodes that were allocated in the filesystem when it was created. Another approach would be to create a “new” filesystem type that is not inode-based. In such a system, a database would hold the information that the inode currently holds. This approach would not be limited by the number of inodes initially created in a filesystem; however, many other parts of the system would be more difficult to implement, and the existing filesystem code could not be leveraged to ease the implementation.

5.2 Performance Features

Given the number of changes between the BUMP system and the TrISH system, it is hard to know which changes are contributing the most to the performance improvements. However, some logical conclusions can be drawn.

On-disk data feature was initially designed to help prevent files from being restored when a user searches through a directory for a particular file. However, in practice when a file is read it is read from beginning to end, making the effectiveness of leaving a small portion of the file in on-disk space dubious. Also, if all migrated files kept the 8 K-bytes of on-disk space required (see section 3.2.3) to prevent false reloads when being accessed with `file` or `head` commands, they could consume a large amount of disk space, especially if there are a large number of migrated files. For instance, if 20,000 migrated files retained 8 K-bytes of data, they would hold 160

M-Bytes of data. This results in the conclusion that filesystem wide nonreleased on-disk data are, most likely, not a good idea. On the other hand, if the few files with a high probability of only having their first 8 K-bytes of data accessed could be identified, this feature would be useful. In practice this is hard to do and should probably be done by a user who knows what the future access patterns of the file will be. There are also certain applications and data formats that could take advantage of the on-disk data feature to allow quick access to rough data or directory information with slower access to more detailed data. For instance image files, account history files, mail logs, and backup files have potential for taking advantage of on-disk data.

The real performance gains from the implementation of on-disk data come from four areas. First, a file can be accessed while it is being restored from offline storage. Second, a file's offline data remain valid until the file is modified, thus preventing unnecessary copies to offline storage. Third, the on-disk data are immediately available to the requesting process without intervention from the reload servers. Fourth, the free space management processes are intelligent about releasing space and retaining on-disk *staging area* data. A few of these will be discussed below.

In the BUMP system, a migrated file's data may reside in a "staging area." When a migrated file that has data in the staging area is accessed, a reload process is created to actually move the data from the staging area to the file. This process is faster than going to offline storage for the data, but it still involves creating a reload process, checking the database, and moving the data to the file. The TrISH system implements the same functionality with on-disk data. This has many benefits. Since on-disk data are already in the file, it is not necessary to create a reload process. This greatly improves the access latency for files with on-disk data. If the migration daemon is unavailable, files with on-disk data can still be accessed as if they were nonmigrated files. In the BUMP system, all migrated files are unaccessible when the migration daemon is unavailable.

When free space is required in the filesystem, the TrISH system releases only enough space to satisfy the immediate need. By doing this, many files keep their on-disk data. In contrast, the BUMP system releases *all* staging area data whenever space is required. In most cases, more data are released than is necessary, and any hope of accessing data from the staging area is gone.

In the BUMP system, whenever a file is restored from offline storage, a new process is created. This reload process mounts the offline media, restores the file, unmounts the offline media, and goes away. In the TrISH system, when a reload server restores a file, it keeps the offline media in the device. When another request for offline data on the same media is received, the media is already mounted and ready to use. If the average time to mount an offline volume is about 6 seconds and 10 migrated files whose offline data reside on the same offline volume are restored, the BUMP system would spend 60 seconds just mounting and unmounting the media. The TrISH system would spend only 6 seconds, a significant improvement.

The database access methods in the BUMP system are extremely slow. For instance, it could take up to 30 or 40 seconds to delete 10 migrated files. In the TrISH system, with its intelligent database access methods, it takes only 1.3 seconds to delete 10 files.

The BUMP system selects files for migration based solely on their badness value. When files are assigned a badness value, their value is compared to a system administrator-defined value. If the file's value is greater, it is migrated. It is a very difficult task for the system administrator to fine-tune this value so that *just enough*, but not *too much*, free space is created. The TrISH system takes a different approach. Rather than using the badness value to define *when* a file is migrated, it is used to define *what order* files are migrated. The system administrator just defines the amount of free space he would like in the filesystem. TrISH migrates files until the free space target is reached. This greatly simplifies the task of defining system parameters and also achieves the real goal of guaranteeing free space in the filesystem.

The examples below show the benefit of intelligent reload servers and the TrISH implementation of the staging area data through on-disk data. The file being accessed is a 36 K-byte text file. The application is `wc`, the UNIX word count utility.

The `csh` built-in command `time` is used to track the amount of time required to execute the command. The third column in the output is the elapsed time used by the `wc` command.

The following is the output from the `ls -l` command, which shows that the file has no on-disk data.

```
127 saaz> ls -l test.03
mrw-rw-r--  1 bytheway    36568 Jun  8 01:25 test.03    [102-13595       0]
```

The file is accessed with the `wc` command. The offline media must be mounted by the reload server and the data reloaded into the filesystem. This takes 12 seconds to complete.

```
128 saaz> time wc test.03
     677     882   36568 test.03
0.1u 0.0s 0:12 1% 2+3k 2+1io 0pf+0w
```

The output of the `ls -l` command shows that the file is still a migrated file and that it has all on-disk data. A file in this same state in the BUMP system would be migrated with its data in the staging area.

```
129 saaz> ls -l test.03
mrw-rw-r--  1 bytheway    36568 Jun  8 01:25 test.03   [102-13595    36568]
```

When the file is read, it is accessed in less than 1 second. The TrISH system simply allows the file to be read as if it was not migrated. The BUMP system would have had to create a reload server to move the data from the staging area to the file.

```
130 saaz> time wc test.03
     677     882   36568 test.03
0.0u 0.0s 0:00 50% 3+3k 10+0io 0pf+0w
```

Using the `trelease` program, the file is released so it has no on-disk data. The file is again accessed with the `wc` command. At this point the reload process still has the offline media mounted; it simply reloads the file. This access took only 1 second. The BUMP system would have taken at least 12 seconds to reload the file, just like the first example above.

```
131 saaz> trelease -keep 0 test.03
132 saaz> time wc test.03
     677     882   36568 test.03
0.0u 0.0s 0:01 7% 3+3k 2+1io 0pf+0w
```

5.3 TrISH Simulation

The TrISH system has been running dependably for over 2 years on a test machine. However, due to extenuating circumstances it was never deployed into production use. As a result, usage and performance statistics are not available for production work loads. To analyze the effectiveness of the TrISH system, I will present simulated TrISH activities that were generated using the TrISH simulator. It should be noted that all of the features discussed in this paper are actually implemented in the TrISH system. Also, statistics presented in other parts of this paper are from the running TrISH system. Only the numbers in this section are simulated.

Using the simulator to assess the value of file migration has some values over using a production system. With the simulator I was able to change filesystem sizes and TrISH parameters like the high, low, and target watermarks and add additional trace values to the simulator as they were identified across multiple simulations. These incremental changes, refinements, and "what if" runs would not have been possible with activity logs from a production system.

The TrISH simulator works by creating a virtual filesystem that is smaller than the real filesystem. It migrates and releases files to fit in the virtual filesystem and then uses filesystem inode dump logs to recreate file activity in the virtual filesystem. The simulator very closely resembles the actual activities of the TrISH system. The results of simulating three different filesystems are presented in this section.

5.3.1 Simulation of `sunset:/home/grad`

The `/home/grad` file system, on the file-server sunset, is a 7 Gigabyte filesystem, with about 5 Gigabytes of used space. It holds the home directories of graduate students in the Department of Computer Science at the University of Utah. To determine the size of the virtual disk, an analysis of the size and number of files in the filesystem was done. Figure 5.1 shows the results.

This graph clearly shows that about 90% of the files consume only 20% of the space. The other 80% of the space in consumed by about 10% of the files. If one migrated all of the big files one could reduce the amount of used space by 80%. However, I would like to migrate just the *old* big files, since they will most likely not be accessed again soon. Figure 5.2 shows the relationship between the age, number, and size of files in the `/home/grad` filesystem.

This shows that there is no real relationship between the age of a file and the size of a file. That is, a large file is just as likely to be created today as it was a few months ago. The TrISH system combines the age and size of the file to come up with the badness value (discussed in section 3.3.4). Files with larger badness values are migrated first. Figure 5.3 shows the relationship between badness value, number, and size of files.

This graph is much steeper, showing that there are relatively few *old large* files.

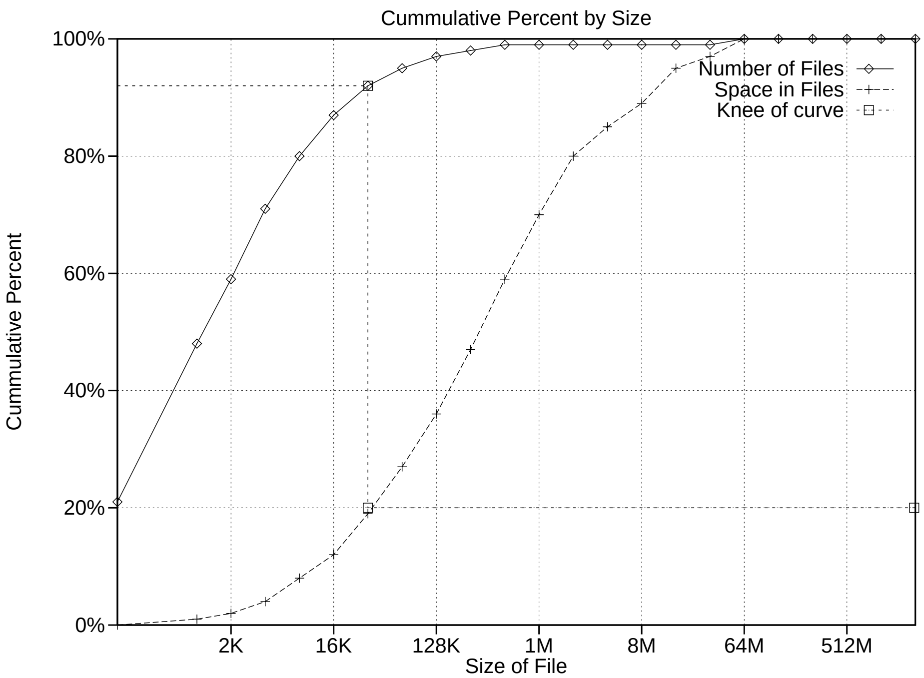

Figure 5.1. sunset:/home/grad – File Size

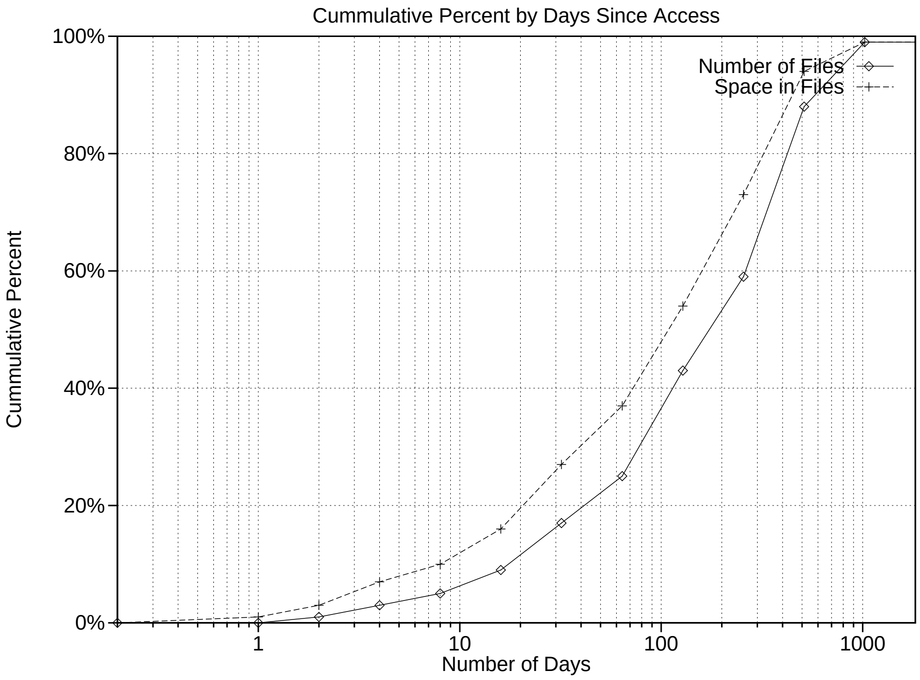

Figure 5.2. sunset:/home/grad – Days Since Access

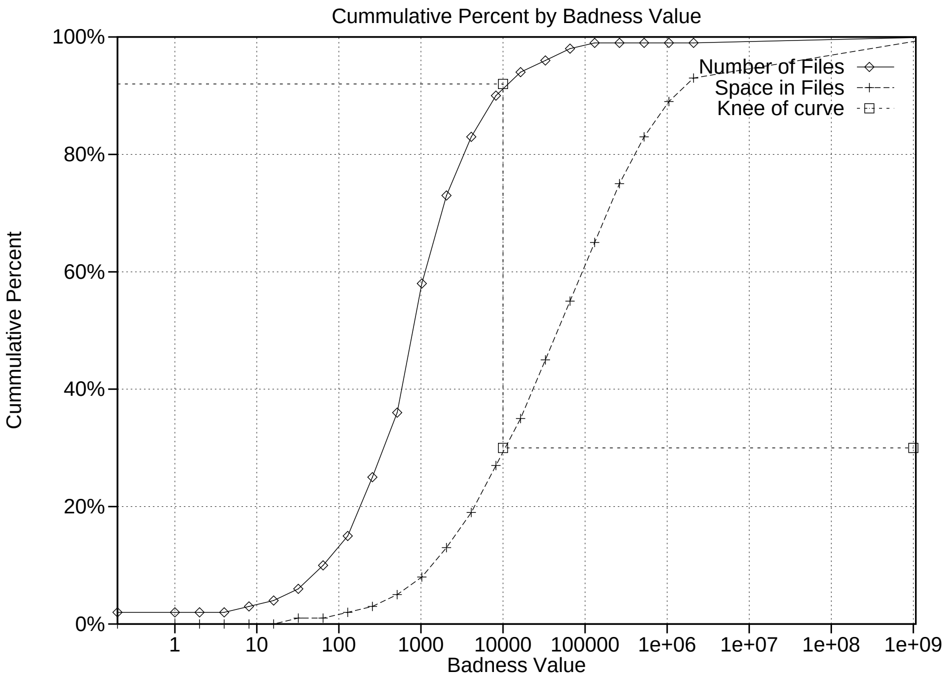

Figure 5.3. sunset:/home/grad – Badness Value

When it comes to migrating files, the fewer that have to be migrated the better. This graph shows that by migrating 10% of the files, one could reduce the filesystem to 30% of its original size. However, this is very close to the edge of the knee in the *Number of Files* curve, meaning that if an unexpected need for space came along, a large number of files would need to be migrated to satisfy the requirement. If the knee of the graph is moved up slightly, then the filesystem can be safely reduce to 35 to 40% of its original size.

The real filesystem has 5 Gigabytes of used space; 40% of this is 2 Gigabytes. Figure 5.4 shows what would have happened to the used space and the nonmigrated space in the filesystem had TrISH been enabled on a 2 Gigabyte filesystem holding these files.

The space between the *Used Space* curve and the *Nonmigrated Space* curve consists of data that have been copied to offline storage and are in the on-disk area of the migrated files. The ability of the TrISH system to keep free space in the filesystem is clearly shown by the *Used Space* curve. When the used space reaches

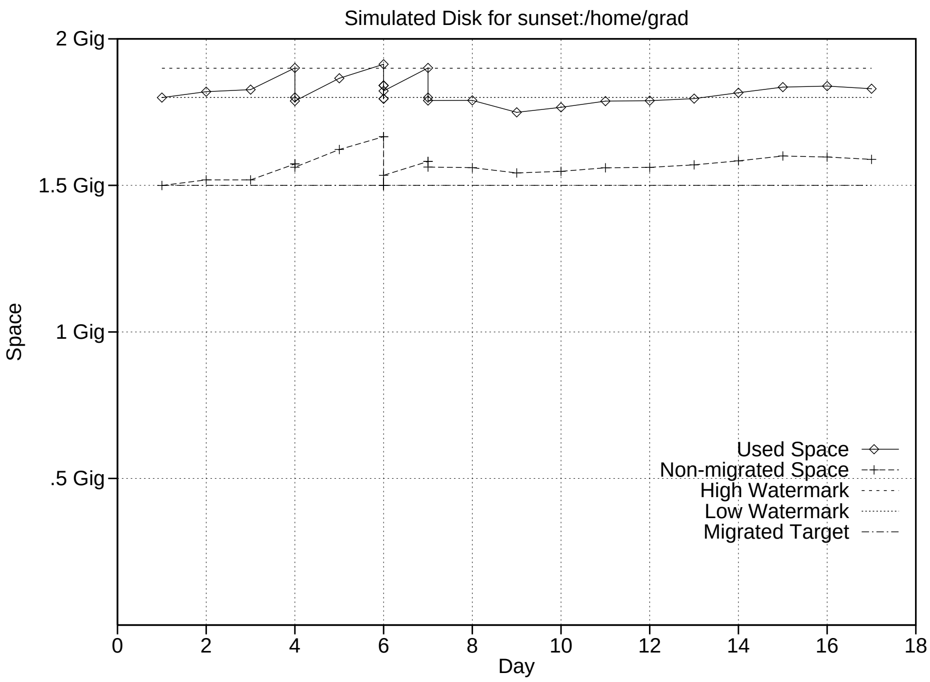

Figure 5.4. sunset:/home/grad – Simulated Disk

the high watermark, the free space creation process converts migrated on-disk space to free space, bringing the amount of free space down to the low watermark. This can be seen on days 4, 6, and 7. When there is no more on-disk space available for free space creation, the outmigration process starts and creates more on-disk migrated space.

To show the impact on the system and to the user, Figure 5.5 plots the total number of files, the number of migrated files, the number of files that were accessed, and the number of accessed files that were migrated and had to be restored or were on-disk and did not need to be restored. A similar graph could be drawn for the amount of space that was migrated, accessed, and restored, but the real impact comes from the number of files, not their size.

The figure shows that, indeed, roughly 10% of the files were migrated. It also shows that between 1 and 4% of the files are accessed on any given day. It shows that on average, only about 50 files had to be restored, with a peak of 200 files on day 7. Also the on-disk space saved a number of restores. On days 4, 5, and 6, it

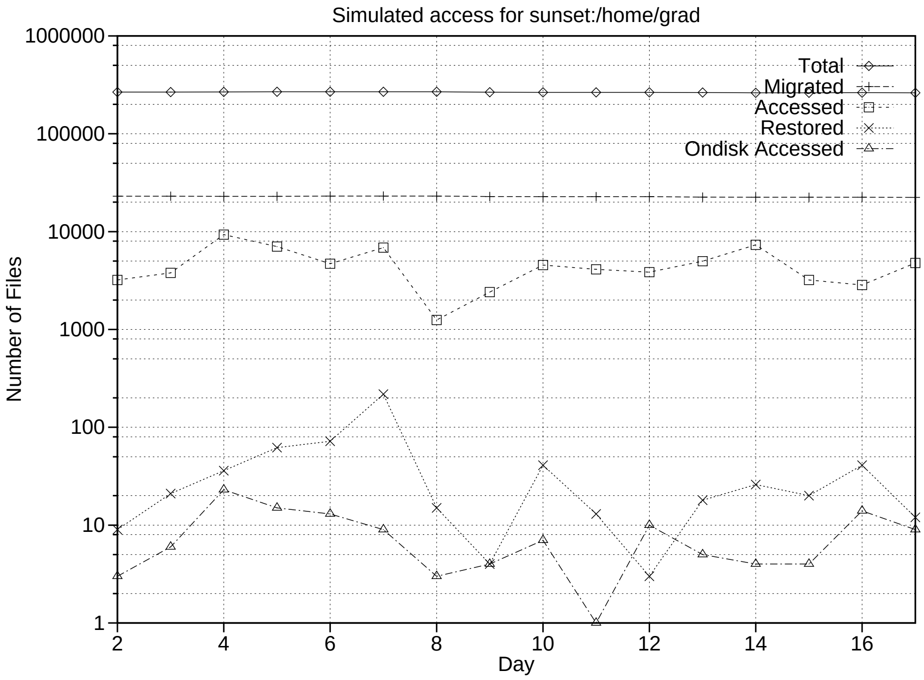

Figure 5.5. sunset:/home/grad – Simulated Access

saved over 10 reloads each day, with day 4 saving 20 reloads. The BUMP system would have had to restore these files from offline media.

By impacting only a very small number of file accesses, the TrISH system is able to squeeze this 5 Gigabytes of used space into a 2 Gigabyte filesystem, saving at least 3 Gigabytes of disk space and possibly more, since free space management is handled by TrISH rather than overallocating the size of the filesystem. This filesystem seems to be a good candidate for the TrISH system.

5.3.2 Simulation of `geronimo:/u`

Geronimo is the file server for the Utah Supercomputing Institute (USI). The `/u` filesystem is the home directory (permanent) space for users of the USI computing facilities. During the following analysis `/u` was an 8 Gigabyte filesystem and was 95% full. About 20 days into the statistics gathering process, the filesystem was rebuilt into a 12 Gigabyte filesystem. This effectively rendered the remaining traces unusable. However, some interesting facts can be observed during the first 20 days of data gathering.

Each of the USI machines has a large filesystem set aside for big temporary files; these files are not usually written into the `/u` filesystem. At various times in the past, some users have had up to 6 Gigabytes of space in these temporary filesystems. Rather than use the UNIX quota system to control space usage in the `/u` filesystem, users are billed for their space at a small monthly rate, giving them some incentive to keep there accounts clean of large dead files.

The same figures that were presented for the `sunset:/home/grad` filesystem will be presented for this filesystem. Figure 5.6 shows that in spite of the incentives for the users to keep their accounts free of large files, this filesystem has a proliferation of very large files. In fact, 5% of the files occupy 90% of the space.

By examining the age distribution of the files, shown in Figure 5.7, one can see that this is a young filesystem, at least compared to the `sunset:/home/grad` filesystem. Very few files have not been accessed in the last year. Like the last filesystem, there is no strong correlation between the size of a file and the age of a file, although there are a number of large files that are about 30 days old.

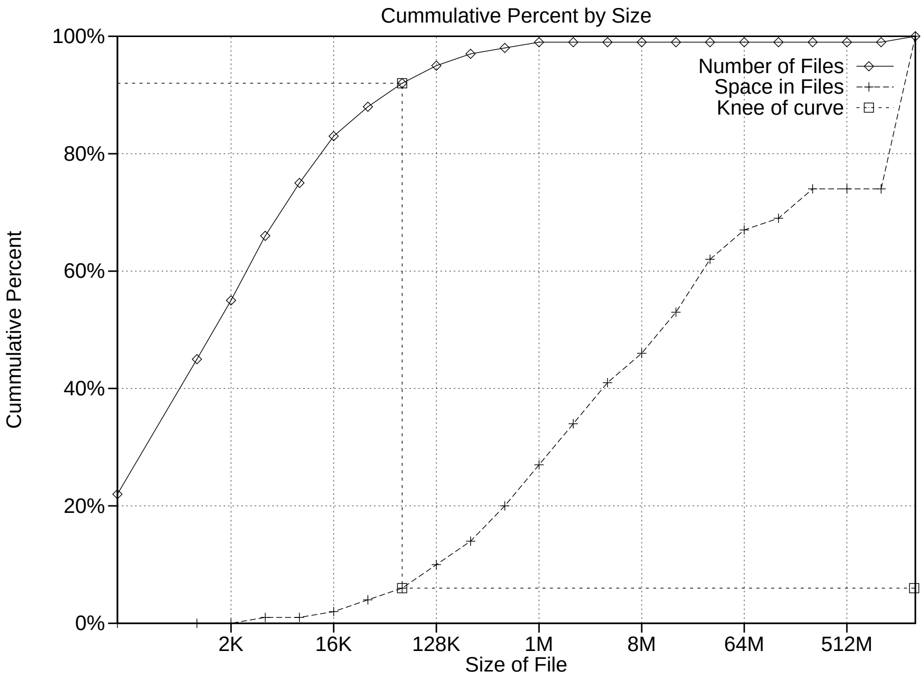

Figure 5.6. geronimo:/u – File Size

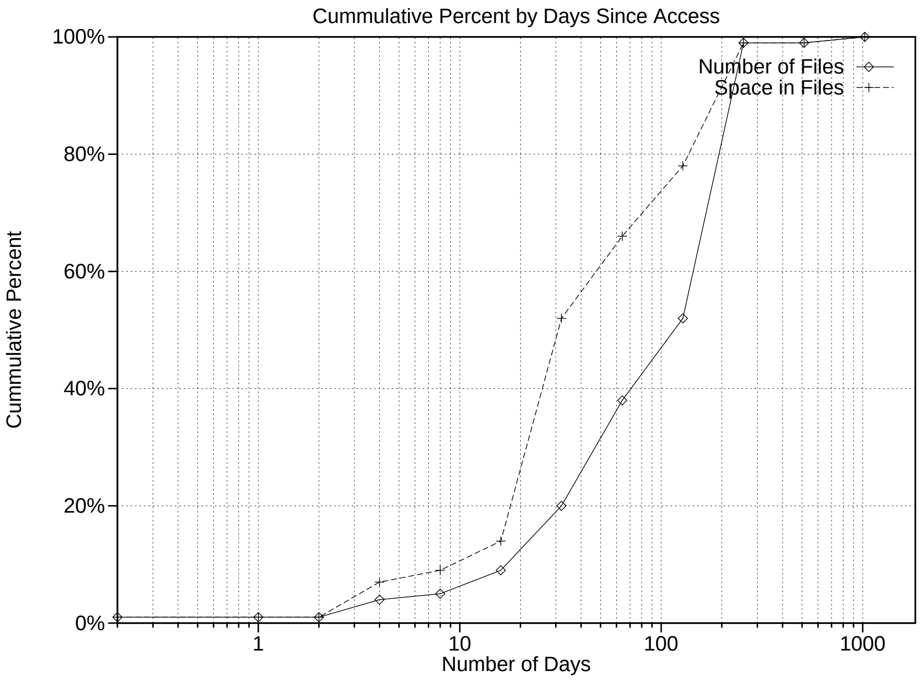

Figure 5.7. geronimo:/u – Days Since Access

By again staying on the upper side of the knee on the *Number of Files* curve, shown in Figure 5.8, it can be observed that 95% of the files, as mapped by the badness value, occupy only between 15 and 20% of the used space. By taking 18% of 8 Gigabytes, a virtual filesystem of 1.5 Gigabytes is calculated.

As shown in Figure 5.9, this filesystem is quite active. Free space creation occurred every few days, and outmigration occurred regularly. In fact, to prevent constant free space creation and outmigration processing, the low watermark was lowered to 85% and the migrated target watermark was lowered to 55% of the virtual filesystem size. With all this filesystem activity, one would expect that there would also be a lot of migrated file reloads.

In fact there are surprisingly few reloads, as shown in Figure 5.10. The number of reloads peaked at about 40, and, happily, the on-disk space prevents almost as many reloads as there were reloads. As predicted, between 5 and 10% of the files were migrated, and like the `/home/grad` filesystem, around 1 and 3% of the files were accessed on any given day.

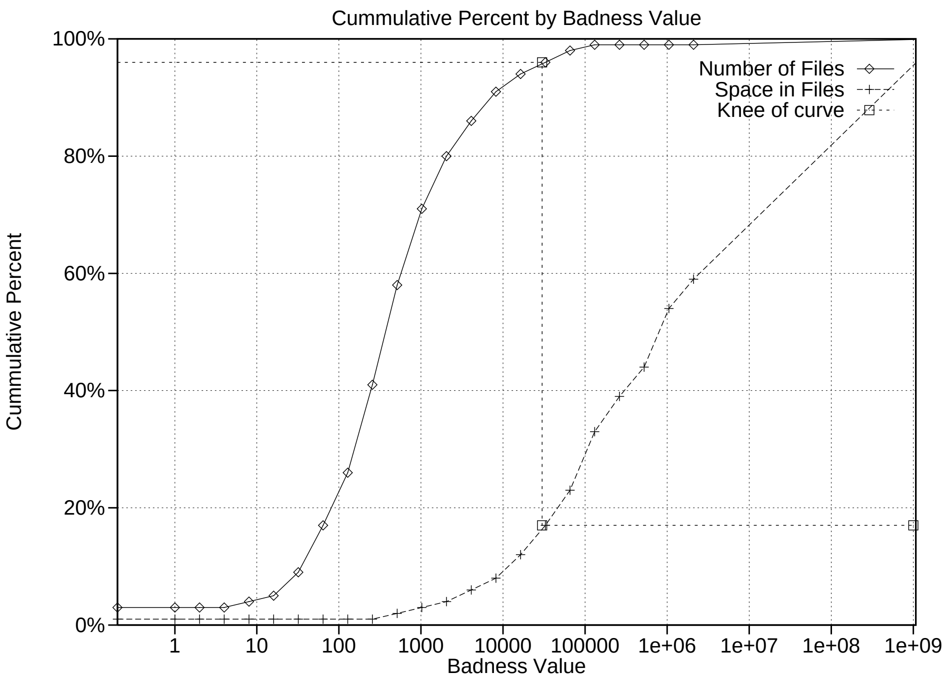

Figure 5.8. geronimo:/u – Badness Value

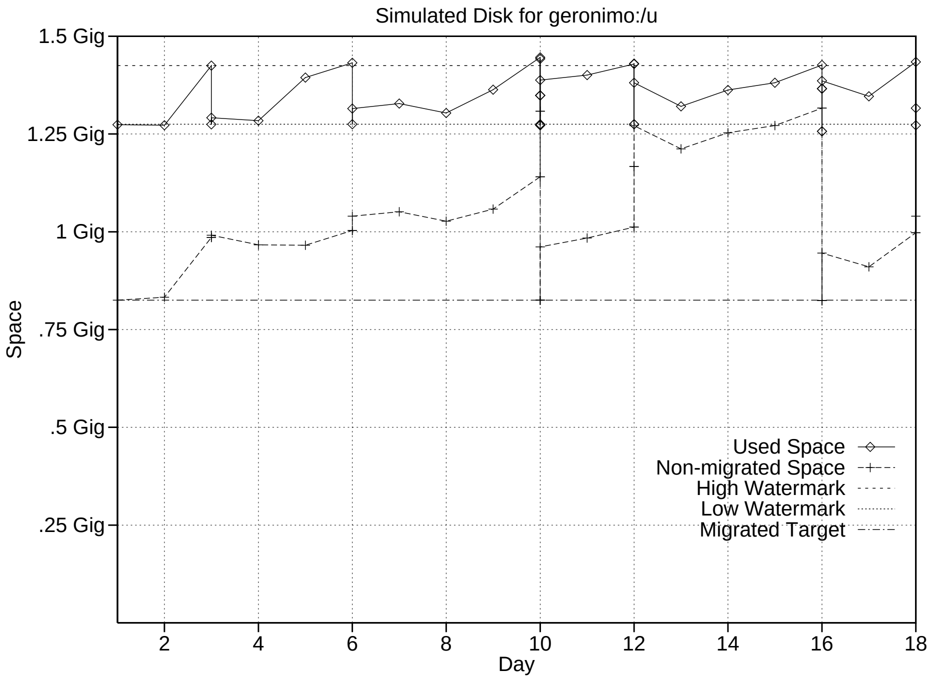

Figure 5.9. geronimo:/u – Simulated Disk

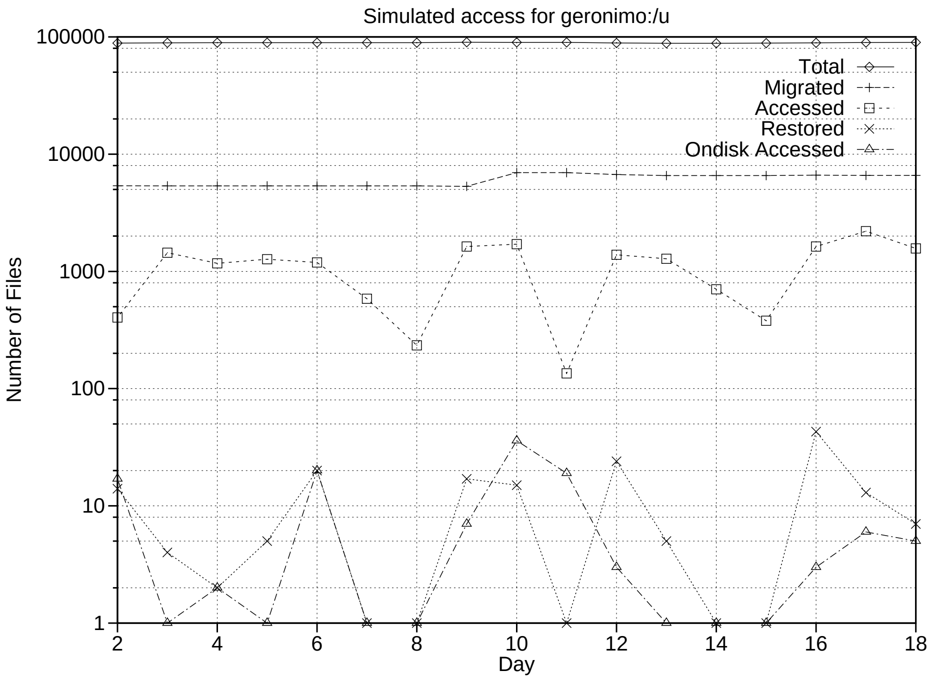

Figure 5.10. geronimo:/u – Simulated Access

This 8 Gigabyte filesystem was easily squashed to a 1.5 Gigabyte filesystem, an 80% savings in disk space usage. Very few files needed to be reloaded, and the on-disk space saved between 50 and 100% of the potential reloads. This filesystem also appears to be a good candidate for the TrISH system.

5.3.3 Simulation of `fast:/usr/lsrc/avalanche`

The `/usr/lsrc/avalanche` filesystem contains the source code for a software project under development in the Computer Systems Laboratory in the Department of Computer Science. It is a 1 Gigabyte filesystem and is about 75% full.

Once again, it appears to have roughly the same file size characteristics as the other filesystems examined. This is shown in Figure 5.11.

Figure 5.12 shows that this is a very young filesystem. Whereas 55 to 65% of the files in the other filesystems have not been accessed in 90 days, only 5% of the files in this filesystem have not been accessed in the last 90 days.

The badness curve in Figure 5.13 shows that about 10% of the files could be

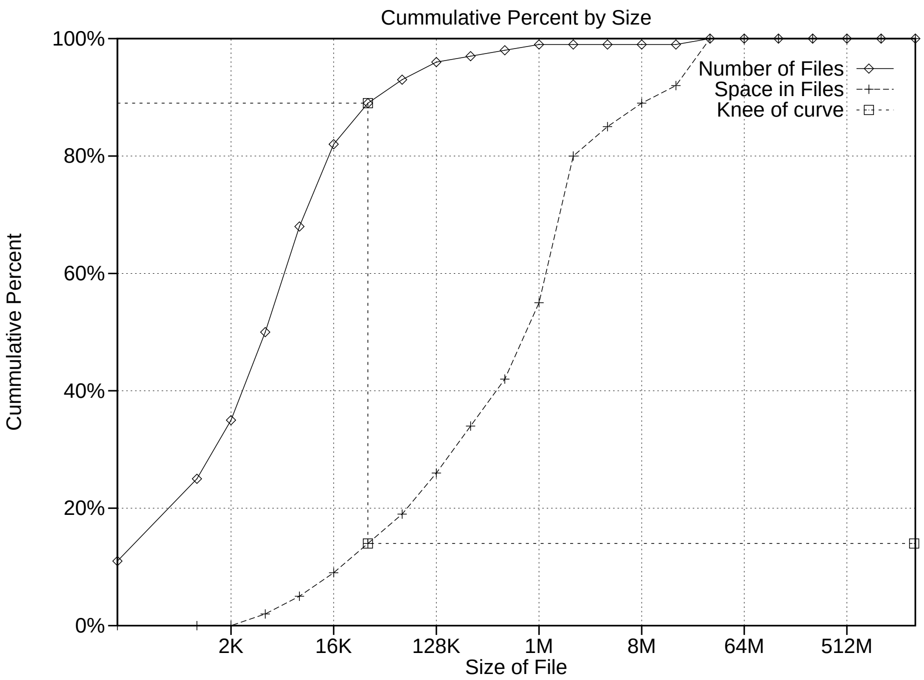

Figure 5.11. fast:/avalanche – File Size

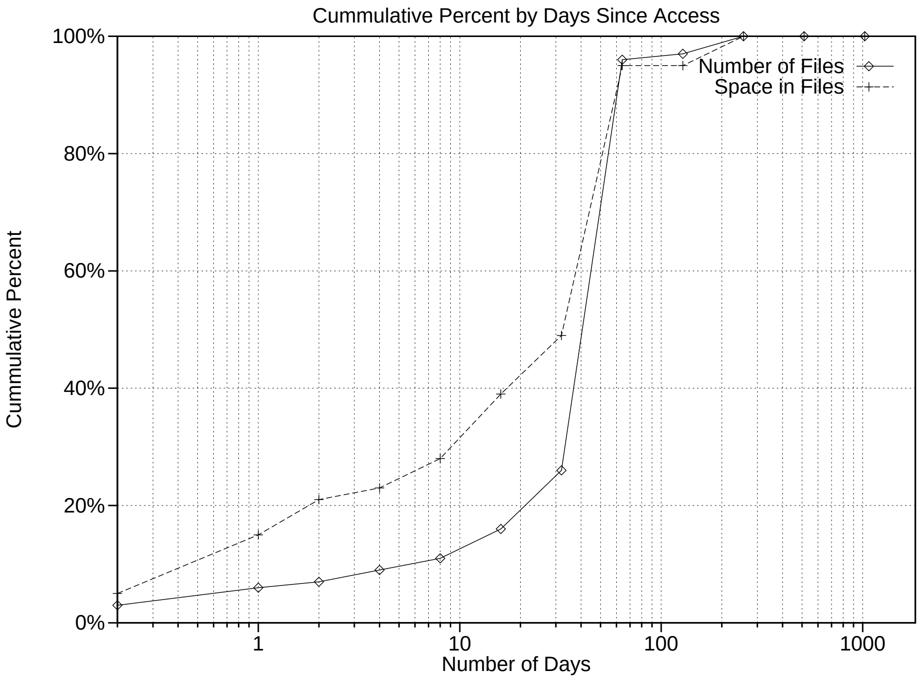

Figure 5.12. fast:/avalanche – Days Since Access

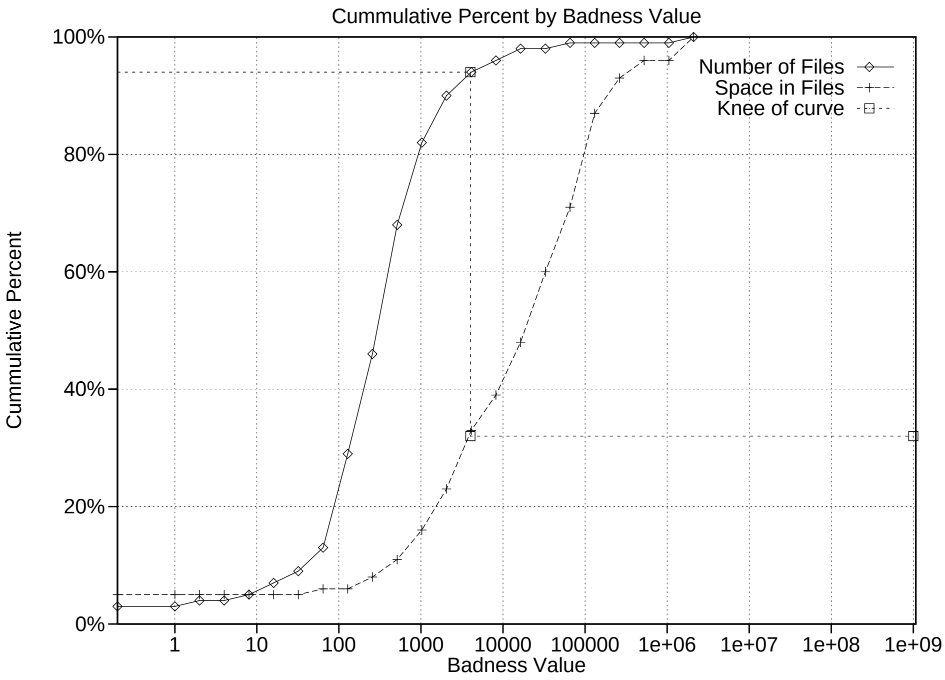

Figure 5.13. fast:/avalanche – Badness Value

migrated, reducing the size of the filesystem to about 35% of its original size. A 350 Megabyte filesystem will hold 35% of this 1 Gigabyte filesystem.

What appeared in this static analysis to be a "so-so" filesystem for migration turned out to be a really bad choice. Figure 5.14 shows that free space creation is occurring many times each day, and out-migration is happening on a frequent basis.

The initial out-migration of files, shown in Figure 5.15, looks like what was expected; about 10% of the files were migrated. Unfortunately, everything went down hill from there. On days 4, 5, and 6, an ever increasing number of files had to be migrated to create the enormous amount of free space required by the new files being created in the filesystem. By day 6, 70% of the files were migrated. On day 7, 1575 files were accessed, and a staggering 1040 of them had to be reloaded. On other days, less than 200 files had to be reloaded, but only 400 to 800 files were accessed on those days.

This is not a good candidate for the TrISH system. It is a young, very active

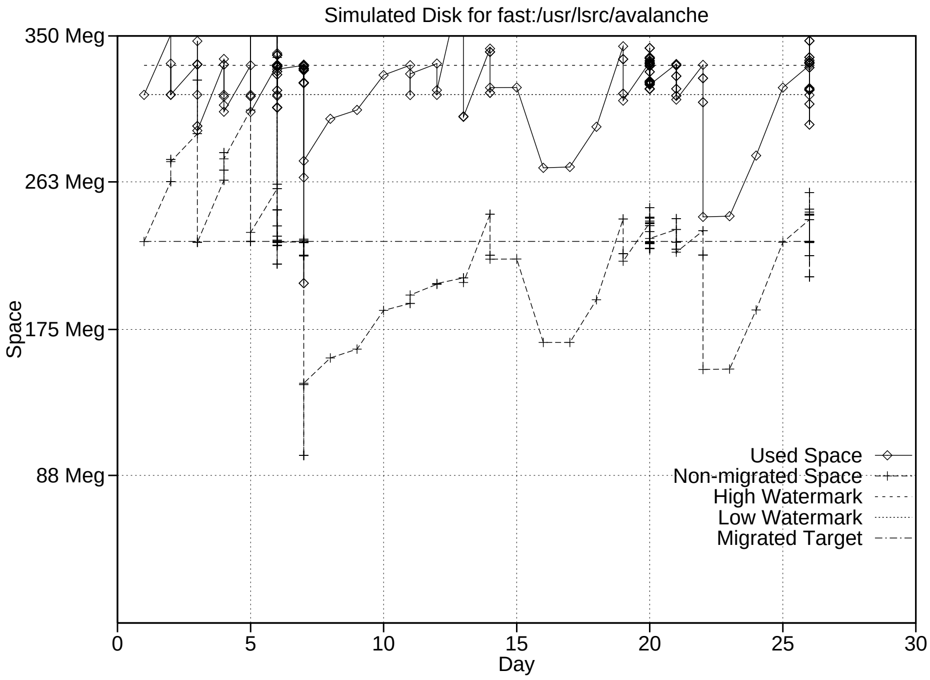

Figure 5.14. fast:/avalanche – Simulated Disk

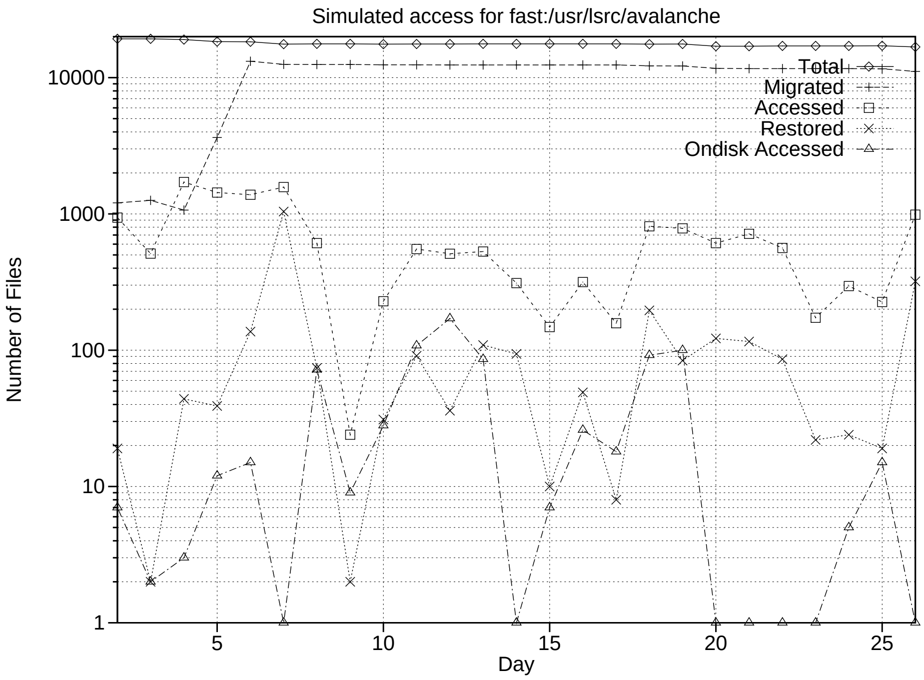

Figure 5.15. fast:/avalanche – Simulated Access

filesystem with no large inactive files to migrate. Also since it started out quite small (1 Gigabyte), the opportunity to save a lot of space did not exist.

This simulation also points out a weakness in the TrISH simulator. The filesystem traces were run nightly. In an active filesystem like this one, a lot of files are created one day and deleted the next. The simulator can only look for deleted files after the entire daily trace file has been processed. Because of this, the simulator will wrongly see that the filesystem is out of space, perform free space creation, finish the trace file, and delete files that did not appear in the trace file. This effect can be seen clearly on the *Simulated Disk* figure for `fast:/usr/lsrc/avalanche` on day 7. TrISH worked very hard to create free space by migrating and releasing files. After the deleted files were removed, the free space in the filesystem dropped down to 63%. The TrISH simulator performs a worst case analysis, which for this filesystem is pretty bad.

5.3.4 Simulation Conclusions

One weakness of the simulator is that it only knows that a file was accessed sometime during the day. It does not know how many times the file was accessed nor what order they were accessed. Since a file's data remain on-disk after they are reloaded, the number of reloads to accesses reported by the simulator is an absolute worst case. It assumes a file is accessed only once. If the true number of file accesses was known, it is very likely that the number of reloads to total file accesses would be a miniscule amount.

These simulations show that a hierarchical storage management system is a viable option for the `sunset:/home/grad` and `geronimo:/u` filesystems. It reduced the disk space requirements of these filesystems and only impacted a small number of files. The `fast:/usr/lsrc/avalanche` filesystem would not benefit from an HSM system, but since it is the source directory for an active development project, one could have predicted this outcome.

5.4 Usability Features

From a user perspective the most important feature of the TrISH system is the ability to actively manage migrated files. A user has the ability to force a file to migrate, to release a file's on-disk data, and to "batch" reload any number of files. He or she also has the ability to prevent the normal automatic file migration process from migrating a file. The user has ultimate control over files and can choose when, where, and why they are migrated.

Users may also be interested in application programs that have the ability to detect migrated files and the amount of on-disk data they hold and to watch for data being restored into them. These applications can provide a more interactive feel to migrated files and allow the user more control over when a file is restored from offline data.

From a system administrator's point of view, the ability to manage the TrISH system is very good, especially compared to BUMP. TrISH has tools to analyze filesystems to determine if they are a good match for being managed by TrISH. The system administrator has the tools to start, stop, drain, and cancel reload servers. He can get kernel-level statistics to determine the impact TrISH has on file access times. New file migration policies are easy to implement, thanks to the modifiable badness value functions and free-space tuning parameters.

Before implementing TrISH on a system, the system administrator can gather statistics on the types of files in the filesystems. If a filesystem looks promising, he can run the TrISH simulator for a few weeks or months to see if the filesystem is actually a good fit for file migration. These preanalysis tools are very important when implementing the system. They help prevent user dissatisfaction by identifying filesystems that are not appropriate for file migration. It is also a useful tool for predicting good starting values for parameters and sizes. Example output from the TrISH simulator is in section 5.3

The TrISH device manager provides a way to manage offline devices and share them between TrISH processes and other processes, such as backups and individual user requests. Through the use of the device manager, TrISH can automatically activate reload servers to meet peak demands and yet return idle devices back to system for use by others. This limits the impact of the TrISH system on the rest of the system. The device manager is very stable and has proven to be very useful. It has been used at the Utah Supercomputing Institute for the last 3 years to manage tape devices for backup purposes.

The TrISH operator messaging system provides a convenient and robust way to communicate with someone when human intervention is required. The X-window system support has proven to be extremely useful and hardy. Even if the X-server hangs or is rebooted, the operator messaging system recovers and is successful at contacting an operator. The operator messaging system has also been used at the Utah Supercomputing Institute for the last 3 years to coordinate the activities of the operations staff.

5.5 What Did Not Work

There was one initial design goal that was dropped because it was technically infeasible. Initially, I wanted to eliminate the need for the file migration identifier.

Both BUMP and TrISH use this field to associate a migrated file's offline data with the file's inode located in the filesystem. This field consumes most of the remaining unused space in the on-disk area of the inode structure. I was hoping that since the inode number is already unique and since the migration database can be stored with the filesystem (eliminating duplicate inode entries between file systems), yet another unique number was unnecessary. This might be the case if migrated files are never restored from backup tapes, but in fact migrated files *are* restored, and when they are, their inode number changes. If the offline data are referenced by the inode number, the restored file will no longer have access to its offline data. Even worse, the migration system might be confused and restore another file's data into the file when it is reloaded. Because of this, the file migration identifier field was kept in the inode.

The informal analysis that I have done leads me to believe that once a file is read, it is read from the beginning to the end. There are a few notable exceptions, such as the `file` and `head` commands, but generally, when a file is read, it is read from beginning to end. In the TrISH simulations I performed, retaining on-disk data usually caused more problems than it solved. The bottom line conclusion is that on-disk data should not be used across the board for all filesystems. Of course, there are a few exceptions to this rule, as was discussed above in section 5.2.

5.6 Future Enhancements

Since every file migration request is passed on to the logging task, there is a potential for performing some intelligent monitoring and processing of events. For instance, it could keep a history of reload requests and automatically de-migrate active migrated files. Doing so would prevent them from being released when space was needed in the filesystem. It could reload all of the files in a directory, if the directory was part of a program source tree. The assumption would be that, if a compile accesses one file, it will most likely access most, if not all, of the files in the same directory.

When a file is updated or deleted, the offline data becomes invalid. When a large number of files have been updated and deleted the offline media becomes full of invalid data. A space reclamation process should be developed. It must be able to handle near-line devices such as optical disk, as well as offline devices such as 8mm tape.

Since backups are performed nightly, the ability to use backup tapes for the offline source would reduce the time spent copying data to offline storage. In general this is a very difficult thing to manage. Since TrISH does not control the backup system or its media, it is not reasonable for it to place any confidence in the offline data or the ability to recover a migrated file using the backup media.

It would be beneficial for an NFS client to have the ability to get migrated file information, such as the amount of on-disk data. The ability to control the actions of the NFS server through process flags would also be useful. In order to provide these capabilites, either the NFS protocol would need to be modified or another ancillary server, similar to `lockd`, would need to be implemented.

5.7 Lines of Code in TrISH

The TrISH system as a whole ended up being much larger than I had anticipated, totaling roughly 47,000 lines of code. Most of the code was in implementing library routines (17,900 lines) and TrISH daemons and utilities (17,200 lines). The operating system modifications required 8,400 lines of code, with most of the code (8,000 lines) being used to implement the communication mechanism between `trishd` and the UNIX kernel. The modified system utilities only required 420 lines.

Although TrISH is a large and complicated system, the bulk of the code as well as the complexity has been implemented in user space daemons and programs rather than the UNIX kernel. Because most of the code is in user space, the task of adding features and solving program bugs is more readily accomplished.

5.8 Conclusions

I have shown that by using a hierarchical storage management system, inexpensive offline devices can be effectively used to store large, inactive files, with minimal impact on the general use of the system. By doing so, space is made available for smaller more active files, ultimately saving money that would otherwise need to be spent on an ever increasing amount of online disk space.

The TrISH hierarchical storage management system provides a well-integrated, well-designed, and easily extendable system for managing the movement of data between online and offline storage.

TrISH has improved on the foundation laid by its predecessor, the BUMP system, in a number of ways. TrISH has improved performance through the implementation of concurrent reload and access, the reuse of offline data, and the redesign of filesystem procedures to eliminate delays to staging area data. It provides improved reliability through the use of distributed data bases and redundant offline copies. It has improved manageability by providing utilities to control the TrISH servers. It is more usable because it allows user controlled migration, reload, and release. In short, it is a robust and effective hierarchical storage management system.

CHAPTER 6

RELATED WORK

Hierarchical storage management (HSM) systems have recently become an area of intense research and development. When this thesis was started, only a small handful of HSM systems existed. Now there are a number of HSM systems, both commercial and research.

In the April 1994 issue of *RS/Magazine*[10], there were 21 commercial HSM vendors listed. Peripheral Strategies, a marketing research company, predicts that by 1998, $120 million will be spent on HSM systems, compared to $3.5 million in 1993[10].

Government and industry are both supporting research on HSM and other storage management systems, such as the "Robo-line" project [2], being conducted at the University of California at Berkeley. Dr. David A. Patterson from Berkeley said "My thesis is that a factor of 1000 increase in storage capacity available on most Ethernets will have a much greater impact than a factor of 1000 increase in processing speed for a gaggle of scientists"[2, p. 1]. Clearly, storage management is, and will continue to be, an interesting area of research and development. In this chapter, three of the HSM systems available at the time this thesis was started, are reviewed and compared to TrISH.

6.1 RASH

The NASA Ames Research Center has developed a UNIX-based mass storage system named "NAStore"—short for *NAS Mass Storage Subsystem*[4]. NAStore includes a file migration system titled *Rapid Access Storage Hierarchy* (RASH).

6.1.1 The Goals of RASH

RASH's purpose, as stated by the authors, was to

> "... transparently and optimally ... manage the use of the various types of storage media in the Mass Storage System. This includes the movement of files between levels of storage; the archiving of files from disk to higher levels; the restoration ... of files ...; and the movement of volumes between the higher levels of the hierarchy. RASH will act to maintain a defined amount of free space on the file systems"[4, p. 16].

Other goals listed for RASH include[5]:

- Ensure that no single media error will result in data loss.
- Provide the ability to turn over to the user the offline media containing the user's data.
- The UNIX kernel modifications should be kept to a minimum and "made as cleanly as possible." However, if there is a conflict between performance and "cleanliness," performance wins.
- Use a commercial database system.
- Archive only regular files. No directories or special files are archived.
- Offline data must be kept on removable media in standard interchange format.

6.1.2 How RASH Accomplished These Goals

To accomplish these goals the RASH system provided a three-level storage hierarchy. Level 0 is magnetic disk, Level 1 is tape cartridges in an auto-loader, and Level 2 is tape cartridges on the shelf. The granularity of data movement between levels of the hierarchy is a file, and a file can exist in only one stage of the hierarchy at a time.

Data are moved between the level 0 (disk) and the level 1 (tape) hierarchies by a set of nightly batch processes. One process identifies the files that can be moved from disk to tape. Another process orders the list, using a system administrator defined formula, and begins moving the files from disk to tape (files at the top of the list are moved first). It continues to move data to tape until the amount of free space in the filesystem reaches a predetermined value, at which point the movement of data to tape stops. If during the day a filesystem becomes full, the list is again consulted and data are moved from disk to tape until the free space value is once more achieved.

When a file is accessed that is not stored on disk, the UNIX kernel sends a message to the RASH daemon process. The daemon forks a process which copies the data from a higher level of storage (tape in an auto-loader, tape on the shelf) to disk. As the data are copied to disk, they are immediately available to the process that was accessing the file. The RASH paper says that if a file spanned tape volumes, the data could be coped from each volume in parallel[5], but no reference is made as to whether this feature was actually implemented.

6.1.3 RASH System Calls

A few system calls were added to the UNIX kernel to support the RASH daemon and system utility programs such as *ls*, *backup*, and *restore*. The added system calls are as follows:

rashopen() Open a file using a *device* and *inode* pair. (A device/inode pair is the information sent from the kernel to the RASH daemon to identify a file that needs to be copied to disk).

rashwrite() Write data into a file that is not completely resident on disk. This system call is used to bypass the normal checks against writing past the disk resident data.

rashclose() Close a file that was opened with rashopen(). Wakes up any process waiting for this file and clears the RASH flags in the inode.

rashcntl() Updates and queries the RASH fields in the inode.

The **rashopen**, **rashwrite** and **rashclose** system calls are only used by the RASH daemon.

6.1.4 RASH UNIX Kernel Modifications

In order to provide this storage hierarchy the UNIX kernel needed to be modified. The routines that were modified include the following:

alloc() If free space drops below a minimum, inform the RASH daemon.

ialloc(), iread(), iupdate() Handle RASH flags in the inode.

iput() If the reference count on an archived file goes to zero (meaning the file has been deleted), inform the RASH daemon.

core() Clear the RASH inode flags and inform the RASH daemon if the core file has been archived.

gethead() Wait for archived files to be completely restored.

rdwr() Delay read if the data being accessed has not yet been restored. Clear the RASH inode flags and inform the RASH daemon of writes to archived files.

copen() Inform the RASH daemon to start the reload process for archived files.

close() If an archived file was modified, delay execution until the restore process has completed.

6.1.5 Similarities Between RASH and TrISH

Some of the goals of RASH are similar to the goals of TrISH. They are as follows:

- Run under the UNIX operating system.
- Provide transparent access to offline storage through the use of the existing UNIX filesystem.
- Ensure data integrity.

- Provide high performance.
- Keep UNIX kernel modifications to a minimum.
- Provide a way for the system to automatically create free space in a full filesystem.

Because of these similarities the TrISH file migration system has many of the features found in RASH. Most notable are the following:

- Allowing data to be available to the user process as soon as they are restored to the file.
- Providing a storage hierarchy.
- Attempting to create free space when the filesystem is full.

6.1.6 Differences Between RASH and TrISH

In spite of these similarities, the goals of TrISH and RASH differ in the following ways:

- TrISH is expected to run on any machine that runs 4.3BSD UNIX. This means any type of offline device could be attached to the system. The TrISH system was designed to accommodate this diverse offline storage pool.
- TrISH cannot assume a common commercial database system will exist on the machine. It was designed so that it can use any database system.
- Where RASH was designed with the intent of providing many tera-bytes of storage, TrISH is designed to provide a more convenient way of accessing offline storage. As such it was designed in an attempt to keep the same interactive feel of online storage. Although there is a slightly different emphasis, TrISH should also be able to provide mass storage capabilities in the multi-Gigabyte range.

- A direct result of the goal to provide a more interactive offline storage system is the added goal that the system must be customizable at the user-level. A user should have the option of modifying the way the migration system handles the reading and writing of migrated data.

- Another subtle point is that RASH seems to be designed for dealing with relatively few very large files, whereas TrISH is designed for large numbers of "small" files. This design will also provide the same level of service as RASH for large files.

- TrISH is not overly concerned about the policies associated with file migration and with offline storage management. These policies will be provided by the system administrator. The TrISH system just provides a mechanism to enforce the policies of the system administrator.

Because of these differences, the TrISH file migration system provides features not found in RASH. Most notable are the following:

- An arbitrary amount of data may be behind in the filesystem when a file is migrated to secondary storage. This will provide a more interactive feel to migrated files.

- Reload servers run constantly, waiting to provide instantaneous restore service, and saving the process startup time for more useful work.

- Database access has been abstracted out to access routines. New access routines can be written to take advantage of any particular database a system may have.

- Offline devices access has been abstracted. When a new offline storage device is added to the system, only the well-defined interface routines need to be written.

- The routines that define policy, like the file migration ordering routine, are well-defined and modifiable by the system administrator.

6.2 UniTree

UniTree is a commercially available mass storage system from the DISCOS Division of General Atomics. It runs on many UNIX and UNIX-based machines including Alliant, Amdahl, CDC Convex, Dec, Fugitsu, IBM and Sun[3].

6.2.1 The Virtual Disk System

The UniTree mass storage system provides what appears to be unlimited disk space to the user. This *virtual disk* is actually implemented as a storage hierarchy that can contain online disk, offline tape, optical disk, tape silos, or other storage devices. Once the user stores data onto the virtual disk, the data move around the storage hierarchy under control of the UniTree system according to site configurable parameters. Data are managed by UniTree only when the data have been explicitly moved to the UniTree virtual disk. UniTree defines the low-level storage hierarchy as online disk space and the high-level storage hierarchy as offline tapes. (This may seem reversed, but UniTree's terminology will be used throughout this section.)

The virtual disk and its storage space can only be accessed through the NFS and FTP network file access protocols. To access the virtual disk through NFS, a client machine must mount the virtual disk into its own file tree using the NFS mount command. Once this is done, users can access data stored on the UniTree virtual disk the same as any other NFS-mounted disk. To access the virtual disk using the FTP protocol, a user executes the FTP command to connect to the UniTree server. He or she then uses the standard FTP commands to store, recall, list, and delete files on the virtual disk.

The UNIX filesystem is not used to store data on UniTree-managed online disk; instead UniTree uses its own storage access mechanisms. This was done to bypass some of the limitations of the UNIX filesystem such as limits on file size, number of files, and filesystem size. The UniTree virtual disk, however, when accessed through NFS or FTP, appears to the user as though it were a regular UNIX file system. All of the normal UNIX filesystem semantics are supported through the NFS access protocol.

When data are first stored on the virtual disk, they are physically stored in online disk space. After a specified amount of time, the data are copied to higher levels in the storage hierarchy. When free space in a given level of the hierarchy drops below a *low watermark*, files are ordered using a site configurable function. Files at the top of the list are purged from that level of the hierarchy until the free space again reaches a *high watermark*. A file is not purged from one level of the hierarchy until a copy exists in a higher level. To improve the speed at which free space can be created, files are regularly copied to higher levels of the hierarchy.

When a file is accessed it must exist on first-level storage (disk). If the data do not exist on first-level storage, they must be retrieved from a higher level of the storage hierarchy. This is true for all file accesses whether they be through NFS or FTP.

Because multiple copies of migrated files can exist in the storage hierarchy and these copies may exist in the same or across multiple levels of the hierarchy, reliability is greatly enhanced. If one level of the storage hierarchy fails, another level can be used to retrieve the data. System databases and directories are *shadowed* on separate disk drives and backed up regularly, also improving the reliability of the system.

Since access to data stored in the UniTree virtual disk can only happen through network protocols (NFS and FTP), file access speed is limited by the speed of the connecting network. However, since UniTree supports many different types of networks, including 100 megabyte per second HIPPI and UltraNet, network speed may or may not be the limiting factor at a given site. File access speed is also limited by the rate at which data can be restored to level one storage. UniTree recommends that files which are heavily accessed should be copied to unmanaged client disk rather than being accessed directly from the UniTree virtual disk through NFS.

A future enhancement to UniTree is the UniTree Client Disk Manager (UCDM).

The UCDM product will manage client machine file systems and automatically move data from the client disks to the UniTree virtual disk. When the user accesses a file that has been moved to the virtual disk, UCDM will transparently restore the file to the client disk from the UniTree virtual disk.

6.2.2 Similarities Between UniTree and TrISH

UniTree and TrISH have a number of similarities. Most notable are the following:

- They both run under the UNIX operating system.
- They provide transparent access to a storage hierarchy.
- They both manage free space for online devices.
- They provide NFS access to data in the storage hierarchy.

6.2.3 Differences Between UniTree and TrISH

In spite of the similarities, there is one significant fundamental difference between UniTree and TrISH; UniTree only provides access to its virtual disk through NFS and FTP, whereas TrISH is integrated into the UNIX filesystem. Some of the implications of this are as follows:

- UniTree provides no automatic relief for out-of-space error conditions in the standard UNIX filesystem.
- A user must explicitly move data to the UniTree virtual disk. As a result, it cannot manage much of the inactive data in a UNIX filesystem.
- When a UniTree stored file is to be accessed, it is copied from offline storage to UniTree's online storage and (per UniTree's recommendations) copied to the local client disk where it will be accessed. During this process three different storage devices are accessed, and the file is read or written five times. The TrISH system, on the other hand, copies data directly from offline storage to online storage where they are accessed. Since TrISH allows the data to be

accessed as soon as they are restored and since it uses the standard UNIX filesystem, the requesting process will very likely find the requested data in the filesystem data cache. If data requests are satisfied from the cache, the data are read or written only twice, a significant performance improvement.

- Even the machine where the UniTree virtual disk resides must use the network access mechanisms (NFS and FTP).

Another difference is that UniTree does not provide a way for the user to modify its behavior, whereas a process running on a UNIX system with TrISH can set various process flags to modify TrISH's behavior.

6.3 DFHSM under MVS

The MVS operating system has a space management system named Data Facility Hierarchical Storage Manager (DFHSM). It provides a number of functions for managing online and offline storage including free space management, data migration through a hierarchy of storage levels, and data availability management. (The relevant differences between the MVS operating system the UNIX operating system are listed in subsection 6.3.2.)

6.3.1 DFHSM Storage Management Functions

DFHSM attempts to provide total online and offline storage management. As such it is very complicated and provides a high-level of integration with other storage management activities, such as backups and restores. Significant planning and administration are required to get the most from DFHSM.

The DFHSM system provides the following functions to the MVS user[7]:

- Free space management for online storage. This is accomplished by releasing unused space from the end of overallocated data sets (files) and deleting data sets that meet criteria defined by the system administrator.

- Migration of eligible data sets. The migration can be from level 1 storage (disk) to level 2 storage (tape) or from level 2 storage to level 1 storage. The migration of data from one level to another is automated by the system; there is no user involvement.

- Restoration of data from level 2 storage to level 1 storage. The restoration occurs through reference of the data set or through user command. The restoration of data from level 2 storage to level 1 storage requires user involvement.

- Data availability by managing disaster backup and recovery functions, automatic physical disk backup and restore functions, and automatic data set backup and restore functions.

The DFHSM system also provides an interactive facility for users to request processing for their data sets. It provides bad tape error recovery, data compression, optimum reblocking of data sets during restore processing and internal control data set reconstruction from journal and checkpoint files[7].

The space management tasks performed by DFHSM can be broken down into the following categories[6]:

- Daily space management
- Space-saving functions
- Automatic recall
- Interval migration
- Command space migration.

The daily space management tasks are performed once a day with the goal of creating enough free space on online storage to handle the day's workload. The tasks performed include the following:

- Delete temporary data sets.
- Release overallocated space in data sets.

- Delete any data set whose explicit expiration date has passed.
- Reduce the number of data set fragments so that the data are stored contiguously on the disk. This is accomplished by migrating then restoring the data set.
- Migrate data sets from disk to tape based on last reference date until the amount of desired free space is achieved.

Most of the space saving functions that are performed by DFHSM are closely tied to the way files are managed in MVS; they include the following:

- Compaction of data written to level 1 and level 2 storage.
- Small data set packing. This involves combining multiple level 0 data sets into one level 1 or 2 data set. If the files are small, this saves space because the size of storage allocation in MVS is about 64 kilobytes.
- Partitioned data set compression. When members of a partitioned data set are modified, the old space where the member used to be becomes unusable. DFHSM makes this unusable space in partitioned data sets usable again.
- Automatic blocking of data for maximum storage capacity. MVS data sets do not have a fixed block size like the UNIX file system. The block size can be manipulated to provide the most efficient use of disk space.

The automatic recall process restores a data set from level 1 or 2 storage to online storage. When a file is referenced, the system catalog is searched. If the volume the data are stored on is "MIGRAT," DFHSM copies the data from offline storage to disk and updates the catalog. The system then allows normal file processing to continue.

The interval migration process deals with out-of-space conditions during system execution. When a predefined minimum free space limit is exceeded, DFHSM begins migrating data to level 1 and level 2 storage. When the minimum free space limit is again reached, migration stops.

Command space migration is the way a user can manually control the otherwise automatic functions of DFHSM. For instance, using a DFHSM command, a user can force the restoration of a data set from level 1 or level 2 storage.

6.3.2 Differences Between MVS and UNIX

There are a number of differences between the UNIX operating system, where TrISH was designed to run, and the MVS operating system, where DFHSM was designed to run.

Most of the interactive work in MVS is done within a transaction processing monitor (TP monitor[1]) which opens all of the files it will need as an initialization step when it first starts. The TP monitor is started in the morning and runs all day. As transactions are processed the files are already open, so data are just read from or written to the file.

This is very different from UNIX where interactive programs open and close files frequently. As a result very few files would need to be restored *on demand* under MVS, whereas UNIX would require a large number of *on demand* restores.

In MVS, files are located by their name, volume label, and unit type. Because this is a lot of information to keep track of, the system provides a central catalog that stores this information. The catalog is indexed by file name, and it stores the volume label and unit type of the media where the file is stored. When a program wants to open a file, it specifies the file name. The operating system uses the catalog to determine the volume and device type. The program does not know *which* disk or tape the data are being retrieved from or even, in the case of sequential data sets, *whether* it is being retrieved from disk or tape. When a file is moved between one volume and another, the catalog is updated with the new volume name and unit type. A program accessing the data need not know where a file is stored since it uses the catalog to find it.

[1]Customer Information Control System (CICS) is the most common TP monitor for MVS.

In UNIX, files are named and located through the filesystem, which only supports online devices. If data are stored offline, the user is responsible for keeping track of the media and the devices that can read it. When data are moved from online to offline storage, all programs that access the data need to be changed to access the data from the offline storage.

Since MVS is a proprietary operating system that runs on proprietary hardware, it can be changed without having to worry about unknown side effects. UNIX is an open operating system that runs on many different kinds and types of hardware and uses many different types of online and offline storage devices. Any features added to UNIX must take into account the fact that they will be used on devices that were not available on the development system.

Files in MVS are statically allocated by telling the system how big the file will be when it is created. If the user misjudges the amount of storage needed to store the data and overallocates storage, the extra space is unavailable for use in other files. In UNIX, data space is given to files on an *as-needed* basis, eliminating overallocation waste. A storage management system in UNIX does not need to deal with this issue.

When a file is created in MVS, an explicit expiration date may be specified which tells the system when the file can be automatically deleted. UNIX does not provide this capability.

In an MVS system, files are generally very large, but they are relatively few in number. In a UNIX system the opposite is true; there are generally large numbers of small files. A migration system designed for UNIX must be able to deal with large numbers of files.

6.3.3 Similarities Between DFHSM and TrISH

DFHSM and TrISH both provide the following features:

- Storage hierarchy containing both online and offline storage.
- Free space management where the system tries to keep the amount of free space on online storage within a range defined by the system administrator.

- Transparent access to the offline storage hierarchy.

6.3.4 Differences Between DFHSM and TrISH

There are many differences between DFHSM and TrISH. The reason for most of these differences is the fact that DFHSM was developed for use under MVS, whereas TrISH was developed for use under UNIX.

- TrISH does not provide any of the MVS specific storage management features found in DFHSM, for instance, partitioned dataset compression, release of overallocated space, deletion of expired files, small file packing, and the reblocking of files.

- Because DFHSM is meant to be a total storage management system, it has been integrated with backup and restore functions. TrISH is not meant to replace the regular UNIX backup and restore processes.

- In DFHSM data sets are migrated from level 0 to level 1 or 2 based solely on the length of time since last reference. In TrISH the algorithm to decide when files are migrated can take into account any number of factors, since it is written by the system administrator.

- DFHSM was designed to work well with a fixed set of offline storage devices, whereas TrISH will be designed to work well with an open-ended set.

APPENDIX A

TRISH CONFIGURATION FILE

The TrISH configuration file is where all configurable values are kept. The TrISH configuration file is extendable so that it can meet the needs of the system. For example, badness functions, defined by the system administrator, store configuration parameters here.

A.1 The Configuration Access Routines

A set of library routines has been provided that allow easily retrieval of information from the configuration file. These routines automatically validate parameter types and parse the parameter list. With these routines the device access methods, the database access routines, the badness functions, or any other part of the system can easily retrieve configuration information from the TrISH configuration file.

Figure A.1 contains the code that retrieves the configuration parameters for the Hewlett-Packard optical disk jukebox access method routines. As can be seen, to add another configuration parameter is a trivial task.

A.2 Example TrISH Configuration File

To facilitate a discussion about the TrISH configuration file parameters and options, a sample configuration file is investigated. The first group of items in the TrISH configuration file is the systemwide parameters that are needed by `trishd` to start everything up. They are as follows:

```
#
# General information.
# --------------------
#
parms:
        type = sysparms
```

```
logger_path =  /usr/local/trish/bin/trlogger
log_file =  /usr/local/trish/trishlog
freer_path =  /usr/local/trish/bin/trfreer
kern_interface = /dev/fmig0
run_dir  =  /usr/local/trish
stop_file =  /usr/local/trish/stop_file
ping_time =  30 minutes
```

logger_path Specifies the location of the TrISH log filter program discussed in section 4.5.1.

log_file Defines the location of the systemwide log file.

freer_path The location of the free space creation program.

kern_interface The name of the kernel communication special file discussed in section 4.2.3.

run_dir The directory where the TrISH daemons and servers should run. If one of the servers crashes, the core file will be written to this directory.

stop_file The location of the global stop file discussed in section B.1.

ping_time The time interval at which `trishd` sends ping requests to the reload servers to verify that they are still alive.

```
static int          mnt_fnd=0, acpath_fnd=0, drnum_fnd=0, dc_fnd=0,
                    cmp_fnd=0, cmpp_fnd=0, ucmpp_fnd=0;
static hpmoac_conf  tmp_conf;

sid_stnzparse_desc desc_list[NUM_DESC] = {
  {"mount_point",   SPD_DSTRING,&mnt_fnd,   &tmp_conf.dhc.mount_point},
  {"changer_path",  SPD_SSTRING,&acpath_fnd,tmp_conf.dhc.changer_path, 80},
  {"drive_number",  SPD_INT,    &drnum_fnd, &tmp_conf.dhc.drive_number},
  {"data_check",    SPD_TF,     &dc_fnd,    &tmp_conf.data_check},
  {"compress",      SPD_TF,     &cmp_fnd,   &tmp_conf.compress},
  {"compress_pgm",  SPD_DSTRING,&cmpp_fnd,  &tmp_conf.compress_path},
  {"uncompress_pgm",SPD_DSTRING,&ucmpp_fnd, &tmp_conf.uncompress_path}
};

/* Parse the stanza into the appropriate variables */
num_unmatched = sid_stnzparse( desc_list, NUM_DESC, pairs,
                               *num_pairs, unmatched );
```

Figure A.1. Sample Code to Retrieve Configuration Parameters

To enable file migration in a filesystem, the filesystem is simply listed in the configuration file with all of its migration parameters. The parameters for filesystem entries are as follows.

```
#
# Migrated Filesystem information.
# --------------------------------
#
/u:
        type = filesystem
        mount_point = /u
        database_dir = /u/TrISH
        filesystem_id = 102
        high_watermark = 95%
        low_watermark = 85%
        releasable_watermark = 50%
        keep_bytes = 4096
        levels = level_cmp, level_hpmo

/gradpub:
        type = filesystem
        mount_point = /gradpub
        database_dir = /gradpub/TrISH
        filesystem_id = 101
        high_watermark = 95%
        low_watermark = 85%
        releasable_watermark = 30%
        keep_bytes = 4096
        levels = level_hpmo
```

mount_point Tells TrISH where to find the filesystem in the filesystem hierarchy.

database_dir Tells where to locate the database that contains the entries for the filesystem.

filesystem_id This is a unique, organization-wide, filesystem identifier. If a filesystem is moved from one machine to another, this identifier will link the filesystem's migrated files to their offline data.

high_watermark When the *used* space in the filesystem goes over this percentage, `trishd` is sent a low space message.

low_watermark The free space creation process stops creating free space when it successfully brings the *used* space in the filesystem below this percentage.

releasable_watermark The migration process stops creating migrated files when the amount of used space less the amount of releasable space in the filesystem falls below this percentage.

keep_bytes This tells the free space creation process how much of a file's space should be left on-disk after it is released.

levels This parameter defines the devices in the filesystem's storage hierarchy. When a file is first migrated, it is copied to one of the first-level devices. Although filesystems can share offline devices, they can only share the media in those devices if they also share the same database.

The filesystem entries point to storage levels. Each storage level is composed of one or more devices. Each device in a level must have access to the same set of offline volumes. The parameters for level entries are as follows:

```
#
# Storage level information.
# --------------------------
#
level_hpmo:
        type = level
        devices = hpmo_1.1, hpmo_1.2
        copies = 2

level_cmp:
        type = level
        devices = comp1, comp2, comp3
        copies = 1
```

devices The name of the devices for this level.

copies The number of offline copies that should be made for this level of the hierarchy. Any number of copies may be made at each level. A file is not made releasable until this number of copies has been successfully created.

In order for a device to be usable by the TrISH system, it must have a device entry in the configuration file. For each device entry, `trishd` will create a reload server. If a device is sharable, it can have multiple device entries. This will cause `trishd` to create multiple reload servers that will access the same offline device.

For example, the automatic compress access methods use a sharable directory in a filesystem and can have multiple reload servers using the same device.

The device configuration entries contain the following information:

```
#
# Device information.
# -------------------
#
hpmo_1.1:
        type = device
        device_type = hpmoac
        device_name = hpmo_box1
        initial_state = STARTED | DEDICATED
        path = /dev/sd2c
        mount_point = /mnt2
        changer_path = /dev/ac0
        drive_number = 2
        data_check = TRUE
        compress = TRUE
        compress_pgm = /usr/ucb/compress
        uncompress_pgm = /usr/ucb/uncompress

hpmo_1.2:
        type = device
        device_type = hpmoac
        device_name = hpmo_box1
        initial_state = STARTED | NON_DEDICATED | ALLOCATE
        idle_time = 3min
        data_check = TRUE
        compress = TRUE
        compress_pgm = /usr/ucb/compress
        uncompress_pgm = /usr/ucb/uncompress


#
# The following three compress devices service the same filesystem.
#
comp1:
        type = device
        device_type = compress
        device_name = fmigcomp
        initial_state = STARTED | DEDICATED | SHARED
        compress_dir = /fmig/TrISH/CMPDIR
        path = /dev/null
```

```
comp2:
        type = device
        device_type = compress
        device_name = fmigcomp
        initial_state = STARTED | NON_DEDICATED | SHARED
        idle_time = 5min
        compress_dir = /fmig/TrISH/CMPDIR
        path = /dev/null

comp3:
        type = device
        device_type = compress
        device_name = fmigcomp
        initial_state = STARTED | NON_DEDICATED | SHARED
        idle_time = 5min
        compress_dir = /fmig/TrISH/CMPDIR
        path = /dev/null
```

device_type This field contains the name of the device-type configuration entry that will be discussed below. This is used primarily to link the device access methods to the device.

device_name When a file is copied to offline storage, the volume and device name are included in the database entry. This parameter specifies the name that should be stored in the database. This name is not unique to this device but rather is shared between all devices that have access to the same set of media. For instance, the Hewlett Packard optical disk jukebox has two drives in it, and they both have access to the same set of optical disk volumes. These two drives will share the same device name.

initial_state This parameter contains flags to indicate the initial state of the device and reload server when `trishd` starts up. The valid flags and their meaning are as follows.

- **STARTED** When `trishd` starts up, it also starts up reload servers for devices whose initial state is `STARTED`.
- **DRAINED** The opposite of `STARTED`.
- **NON_DEDICATED** Nondedicated devices are initially placed in a dormant state. When `trishd` receives more reload requests than it has reload servers, it automatically starts a dormant reload server and starts sending it requests. When the server has been inactive for a short amount of time (defined by the `idle_time` parameter discussed below), it is automatically shutdown by `trishd` and again placed in the dormant state.

DEDICATED Opposite of the `NON_DEDICATED` state. The reload servers for `DEDICATED` devices will not be automatically shutdown, rather they must be explicitly shutdown by the system administrator.

SHARED Most offline devices can only be used by one processes at a time. `SHARED` devices, however, can be used by multiple processes at the same time.

ALLOCATE Devices marked `ALLOCATE` are allocated using the simple device management system supplied with TrISH. Allocated device entries do not need to specify many parameters that non-allocated devices must specify, for instance, the name of the device special file.

idle_time This is the amount of time this device should be idle before it is automatically shutdown and placed in a dormant state. This parameter is required only if the device is `NON_DEDICATED`.

path This is the path name for the device special file. This parameter is required for devices that are not dynamically allocated.

The other parameters in the example configuration file above are device-specific parameters. Some of these parameters are only required if the device is not dynamically allocated with the device manager since that information is returned by the dynamic allocation routines. The Hewlett Packard optical disk jukebox parameters are as follows.

mount_point Optical disks are formated with the regular UNIX filesystem and must be *mounted* in the directory tree to be accessed. The `mount_point` parameter specifies the location to mount this device. This parameter is required only for nonallocated devices.

changer_path This parameter designates the path of the robotic control mechanism inside the jukebox. This is required for nonallocated devices.

drive_number The drive number, along with the `changer_path`, is used to move optical disks between the storage slots and the drive. This is only required for nonallocated devices.

data_check If set to TRUE, the access methods generate CRC codes to verify the validity of the offline data.

compress If set to TRUE, the access methods automatically compress the data as it is being written to the optical disk.

compress_pgm This is the path name of the compress program. This is required only for devices that are doing compression.

uncompress_pgm This is the path name of the uncompress program. This is required only for devices that are doing compression.

Devices of the same type are accessed using the same set of access method routines. The parameters that configure the behavior of the access method routines are defined in the `device_type` entries. In the example below, the `compress` device type has special parameters similar to the ones discussed above. The device-type parameters are as follows.

```
#
# Device type information.
# ------------------------
#
#
hpmoac:
        type = device_type
        media_type = mo_disks
        access_method = hpmoac
        reloader = /usr/local/trish/bin/treloader
        block_size = 8K
        granual_size = 5Meg
        capacity = 768Meg
        mount_time = 7sec
        data_rate = 1Meg
        ir_gap = 0
        cost = 100

compress:
        type = device_type
        media_type = NONE
        access_method = compress
        compress_pgm = /usr/ucb/compress
        uncompress_pgm = /usr/ucb/uncompress
```

```
reloader = /usr/local/trish/bin/treloader
block_size = 8K
granual_size = 2Meg
mount_time = 0
data_rate = 256Mb
ir_gap = 0
cost = 500
capacity = 0
```

media_type Defines the type of media used in the device. This links the parameters for the media to the device type. The media parameters are overridden by the `device_type` parameters.

access_method Defines the access method routines that are used to access this type of device.

reloader Specifies the path of the reload server program for this device. Currently there is only one reload program which handles all devices. However, if a special reload server was needed for a particular device, it would be specified here.

block_size The block size used to `read()` and `write()` data blocks to the device. Defined to optimize throughput to the device and optimize the utilization of the media.

granual_size Defines the size of granules for this device. Defined to optimize the trade-off between media space utilization, and error recovery and volume handling.

capacity Defines the capacity of the device. When a volume runs out-of-space another volume is used instead.

mount_time Designates the average amount of time necessary to mount a volume. This parameter is currently ignored, but could be used to help `trishd` chose the faster of two reload options.

data_rate Designates the average data rate for the device. This parameter is not currently being used, but was intended to be used similar to `mount_time`.

ir_gap The amount of data lost between separate files on the device.

cost Used to categorize the cost of the storage space. Currently not used, but is meant to help determine which files should be moved through the storage hierarchy.

The `media_type` configuration entries are used to define general parameters for each media type. This information is overridden by the parameters contained in the `device_type` parameters described above.

```
#
# Media type information.
# -----------------------
#
mo_disks:
        type = media_type
        block_size = 1024
        granual_size = 512Meg
        capacity = 567K

NONE:
        type = media_type
        block_size = 0
        granual_size = 0
        capacity = 0
```

Using the TrISH control program (`trctl`), the system administrator and system operators manages the activities of `trishd`, the reload servers and reload requests. The `user` configuration parameters designate who the system administrators and operators are and the authority level they have been granted.

```
#
# User information.
# -----------------
#
root:
        type = user
        user_name = root
        authority = superuser

bytheway:
        type = user
        user_name = bytheway
        authority = superuser

lepreau:
        type = user
        user_name = lepreau
        authority = superuser

mike:
        type = user
        user_name = mike
```

```
        authority = system

stoller:
        type = user
        user_name = stoller
        authority = operator
```

user_name The userid of the user.

authority One of three levels of authority to be granted to the user. The levels of authority are:

- **operator** Users with operator authority can start and stop reload servers and cancel outstanding reload requests.
- **system** Users with system authority can do anything operators can do as well as shutdown the TrISH migration system.
- **superuser** Superuser authority is currently the same as system security, but this level was defined for future use.

APPENDIX B

THE OUTMIGRATION PROCESSES

As was introduced in section 4.1, the steps necessary to migrate a file are identify the files to migrate, migrate them, copy their data to offline storage, mark them as releasable, and release their space when needed. In this appendix, these processes will be examined more closely.

B.1 The `tridentify` Process

The first step to migrating files is to identify which files are eligible to be migrated. The `tridentify` program does this in the TrISH system. The `tridentify` program saves a copy of the out-migration database, deletes the database, and recreates a new, empty one. It then executes the following commands to load the new database.

```
find $FS -xdev -type f -print | trstop | troutdbload $FS
```

The `find` command locates and prints all nonmigrated files in the filesystem. This list of files is filtered by the `trstop` program, and the final list of files is assigned badness values and loaded into the out-migration database by the `troutdbload` program.

The `trstop` program is used to stop files from being migrated that should not be migrated. For instance a user's dot files (`.cshrc`, `.login`, `.forward`) the programs needed by the TrISH system to function, and the TrISH database files should not be migrated. These file names are placed in the systemwide "stop file." Below is an example of a systemwide stop file.

```
#
# Don't migrate "dot" files (.login, .logout, .plan, etc.)
```

```
#
^.*/\.trish_stop$
^.*/\.[a-zA-Z0-9_\.-]+$
#
# Don't migrate the files TrISH needs to operate.
#
^/usr/ucb/compress$
^/usr/ucb/uncompress$
^/usr/local/trish/.*$
#
# Don't migrate the TrISH migration database files.
#
^.*/TrISH/.*$
^.*/trishdb.*$
^.*/granual.*$
^.*/volume.*$
^.*/inprocess.*$
^.*/nextfid.*$
^.*/currvol.*$
^.*/outmigdb.*$
^.*/migratable.*$
^.*/releasable.*$
```

In addition to the global stop file, users can have their own personal stop file in their home directory, named `.trish_stop`. The file names contained in it are prevented from being migrated. This feature has obvious advantages for a person who uses a file frequently enough that he does not want it to be migrated but infrequently enough that the migration system would migrate it anyway.

B.2 The `trmigrate` Process

Converting regular files into migrated files is the next step in the migration process. When `trmigrate` selects a file to be migrated, it is added to the inprocess database and made into a nonreleasable migrated file using the `fmig_migrate()` system call. (An "inprocess" file is a migrated file that has not yet been copied to offline storage.)

The `trmigrate` program calculates a "target" value by multiplying the total blocks in the filesystem by the releasable watermark value and subtracting the free space, the space occupied by releasable files, and the space occupied by inprocess migrated files. If the target value has not been reached yet, it proceeds to migrate the file with the largest badness value. The size of the file is subtracted from the target value. This process continues until the target is reached or until no more migratable files exist.

B.3 The `trcopyout` Process

The migrated files are copied to offline storage by the `trcopyout` program. There may be multiple levels in the storage hierarchy, but at this point in the migration process, the files are only copied to the first-level of the hierarchy. However, the `trcopyout` program can be configured to make multiple copies of the file on the first-level device.

The `trcopyout` process finds an available first-level storage device. Even though this device will be the only one used by `trcopyout`, if the device supports removable media, like tape drives, multiple volumes may be used.

The first file from the in-process database is selected, and a granual is allocated using the device's granual allocation routine. The target volume is mounted if the device has removable media, and the device and file are opened. Data are copied from the file to the granual, until the granual is full. If another granual is needed, it is allocated and its volume is mounted. This continues until the file has been totally copied to offline storage. The copy counter in the in-process record is incremented.

When no more copies of the file need to be made, the record is deleted from the in-process database and the file is marked releasable using the `fmig_releasable()` system call. Furthermore, files that were forced to migrate with the `trforce` command are released using the `fmig_release()` system call. An entry is made in the releasable database so that the free space creation process will know the file's space is eligible to be released.

The next file is selected and the process continues until the files are all copied to offline storage.

APPENDIX C

FREE SPACE CREATION

When space in the filesystem is low, the kernel sends a message to the migration daemon notifying it of the low space condition. The migration daemon starts the `trfreer` process, which is responsible for alleviating the space problems.

One activity that the `tridentify` process performs that was not discussed in section B.1 is the loading of the releasable database. This database contains the names and badness values of migrated files with releasable on-disk space.

The `trfreer` process first calculates a target free-space amount by multiplying the size of the filesystem with the filesystem's low watermark and then subtracting this value from the amount of used space in the filesystem.

The file with the highest badness value is selected from the releasable database and its space is released with the `fmig_release()` system call. The amount of space released is subtracted from the target, and the process continues until the target has been met or until there are no more releasable files.

After the first file is released, the `trfreer` process sends a message to the migration daemon informing it that it was able to create some space. The migration daemon, in turn, sends a message to the kernel that free-space was created in the filesystem and that any processes which are blocked waiting for space should now be allowed to run.

A user can force a file's space to be released by using the `trelease` command. This command verifies that the user has permissions on the file and then calls the `fmig_release()` system call. If the user owns the file or has read access to it, then he has sufficient authority to release it.

APPENDIX D

THE TRISH RELOADER SERVERS

The TrISH reload servers are responsible for listening for and responding to control messages from `trishd`. These messages include reload requests, cancel reload requests, status update requests, shutdown requests, and regular "pings" to make sure that the reload server is still alive. The reload server must be able to accept requests at any time, even when it is busy responding to a previous request. For instance, a *cancel* request could arrive while the reloader is busy restoring a file. The list of requests that a reloader must deal with are in Table D.1.

When a *reload* request is received, the server can either accept or reject it. A request is rejected if the server is currently in the process of reloading a file or if the necessary volume is busy. When a request is rejected, `trishd` will requeue the request and may, at a later time, send it back to this reloader again. Accepted reload requests are sometimes *failed* by the reloader because an unrecoverable error has occurred. Failed reload requests are also requeued by `trishd`, except that the request will not be sent to this server again. A reload request will fail only when all of the granuals containing data for the file are unaccessable and the file cannot be fully restored.

Table D.1. Requests for TrISH Reload Servers

Request	*Action To Perform*
`RELOAD_IPC_RELOAD`	Reload a file from my device
`RELOAD_IPC_CANCEL`	Cancel current reload request
`RELOAD_IPC_SHUTDOWN`	Shutdown after current reload finishes
`RELOAD_IPC_PING`	Return *PONG* to verify I'm alive
`RELOAD_IPC_STATUS`	Send status information to `trishd`

The *cancel* request will cause the reload server to abort the current reload request. The *ping* request will cause the reload server to return a "pong" so that `trishd` will know that this server is still alive.

The *status* request will cause the server to return updated status information. Status information is sent to `trishd` when something interesting changes. For instance, when the reloader mounts a new offline volume, a status update is sent.

Graceful recovery from device, database, and media errors is a key feature of the TrISH reload servers. If an error occurs while reading a granual, the reload server will search the database for another granual on an accessible volume. The reload is restarted from where the error occurred using a new granual. If all offline granuals are unreadable or if the device fails, the server notifies `trishd` that the request failed. The request can then be sent to another reload server. While restoring a file, the reload server may get an out-of-space error condition. When this occurs, the reload server waits for about 10 seconds and tries to write the data again. After about 12 tries, the reload server gives up and returns a *failed* message to `trishd`.

APPENDIX E

TRISH OPERATING SYSTEM CALLS

In order to set migration system parameters, set process flags, and create, manage, and reload migrated files, a few system calls were added to the operating sytem. They are as follows.

fmig_stat() Returns, in addition to the file status information returned by the `stat()` system call, information about migrated files. Migrated files *look* like regular files to unmodified programs. When a program needs to know if a file is migrated, it can use this system call instead of the `stat()` system call.

fmig_lstat() Returns, in addition to the file status information returned by the `lstat()` system call, information about migrated files.

fmig_fstat() Returns, in addition to the file status information returned by the `lstat()` system call, information about migrated files.

fmig_migrate() Converts a regular file into a migrated file by changing its type to `IFMIG`, setting the on-disk space field to the size of the file, and placing the `fmigid` value in the inode.

fmig_demigrate() Converts a migrated file into a regular file by changing its type to `IFREG` and clearing the on-disk and `fmigid` fields.

fmig_releasable() Marks a file as releasable, meaning that its on-disk space can be released. This is only done after a file's data have been successfully copied to offline storage.

fmig_release() Releases a migrated file's on-disk data. This can only be done if the file is releaseable and has no modified data. If specified, a small amount of data at the begining of the file will be left "on-disk."

fmig_frelease() Releases a migrated file's on-disk data. This system call is similar to the `fmig_release()` system call, except the argument is a file handle rather than a file name.

fmig_open() Opens a migrated file for output. When a reload request is sent from the kernel to the migration daemon, the file's device and inode numbers are sent along in the request. The reload process uses those values, along with this system call, to open the file. The regular `open()` system call cannot be used because the migration daemon does not know the name of the file that it should restore. It only knows the `fmigid`, the device, and the inode number.

fmig_write() Writes data to a migrated file. The standard `write()` system call does not allow data to be written past the current on-disk data. This however, is exactly what the reload process must do to restore a file's data. This system call is used by the reload process to write to migrated files. When data are written to the file the processes that are blocked waiting for data are woken up.

fmig_lseek() Repositions the offset of the file descriptor to a specfied location in the file. This system call is just like `lseek()`, except that it performs the added step of setting `i_ondisk` to the "seeked to" position if it is greater than the current value of `i_ondisk`. This function is used by the reload process to restore "holes" in files when they are reloaded. Without this call, files that originally had holes in them would instead have data blocks where the holes used to be.

fmig_close() Closes a migrated file. This system call is also used by the reload process. It is like the regular `close()` system call, except that it wakes up processes that are blocked waiting for this file to be restored. It also calls `fmig_demigrate()` for files that are fully restored and have modified data in them.

fmig_sethwm() Sets the high watermark for a filesystem.

fmig_gethwm() Retrieves the value of the high watermark for a filesystem.

fmig_setflag() Sets the process flags discussed in section 4.2.2

fmig_getflag() Gets the current value of the process flags discussed in section 4.2.2

In order to provide access to more information about migrated files, the following standard operating system calls were enhanced.

select() The `select()` system call is used to determine if a file is ready to be read from or written to. Select usually only supports character special files, like terminals, printers, and sockets, but it has been enhanced to support migrated files. When a migrated file has data ready to be read, the file is "selected."

ioctl() When used on regular files, the `ioctl()` system call with the `FIONREAD` command returns the number of bytes left to be read from the file. A new command `FIONDREAD` will return the number of on-disk bytes available to be read from the file. By using this call, a program can determine how much data it can request without being blocked.

APPENDIX F

KERNEL TO DAEMON MESSAGES

The migration messages are of fixed size and are easily decoded using the `fmig_msg` structure. The items included in the structure are listed in Figure F.1. A few of these fields are worth discussing.

The `ms_op` field defines the requested operation. The list of possible operations and a short description of their meaning is shown in Table F.1. The last two entries, `FMIG_D2K_DONE` and `FMIG_D2K_FAIL`, are for messages sent from the migration daemon to the kernel; the others are for messages sent from the kernel to the migration daemon.

Most messages are about migrated files. Migrated files are identified by their `fmigid`. The `fmigid` is the constant value that links a migrated file to its data. This value is sent to the migration daemon in the `ms_fmigid` field. Using the `fmigid`, the migration system can identify a file's database entries and find its data in offline

```
struct fmig_msg {
    int                ms_magic;   /* FMIG_MSG_MAGIC */
    u_long             ms_id;      /* ID of this message */
    int                ms_op;      /* operation */
    uid_t              ms_uid;     /* UID of process */
    int                ms_pid;     /* PID of process */
    int                ms_result;  /* may contain errno */
    long               ms_data1;   /* misc data slot 1 */
    long               ms_data2;   /* misc data slot 2 */
    fmigid_t           ms_fmigid;  /* file migration id */
    struct fmig_devino ms_devino;  /* pass to fmig_open() */
}
```

Figure F.1. Contents of `fmig_msg`

Table F.1. Valid Operations for `fmig_msg`

Operation Code	*Description*
`FMIG_K2D_RELOAD_BLOCK`	Blocking reload of file
`FMIG_K2D_RELOAD_ASYNC`	Asynchronous reload of file
`FMIG_K2D_CANCEL_RELOAD`	Cancel reload of file
`FMIG_K2D_UNLINK`	File was unlinked (deleted)
`FMIG_K2D_TRUNC`	File was truncated
`FMIG_K2D_DEMIGRATED`	File was *de-migrated* by the kernel
`FMIG_K2D_OPENING`	File is being opened
`FMIG_K2D_CLOSING`	File is being closed
`FMIG_K2D_CHOWN`	File has a new `UID` or `GID`
`FMIG_K2D_LOWSPACE`	Low space condition in filesystem
`FMIG_K2D_NOSPACE`	No space condition in filesystem
`FMIG_K2D_CANCEL_NOSPACE`	Cancel no space condition
`FMIG_D2K_DONE`	Request to migration daemon successful
`FMIG_D2K_FAIL`	Request to migration daemon failed

storage.

When the time comes to send a reload request to the migration daemon, the name of the file is no longer known. To identify the file in such a way that it can be opened, the `ms_devino` structure is sent to the daemon with the reload request. Using this structure and the new system call `fmig_open()` discussed in Appendix E, the restore process can open the migrated file and reload its data.

The optional fields `ms_data1` and `ms_data2` are only used by a few of the operations. For instance, the `FMIG_K2D_CHOWN` operation places the new userid (UID) and groupid (GID) into these fields.

APPENDIX G

DEVICE ACCESS METHOD ROUTINES

The TrISH access routines and a short description of their function are listed in Table G.1. These routines can be grouped into the following categories: configuration functions, initialization and cleanup functions, volume handling functions, granule handling functions, and data block functions. Some functions, if they are not needed or are not applicable for the device, do not need to be defined. For instance, the *compress* method does not mount or unmount any media and does not have those functions defined.

The configuration functions retrieve configuration parameters for the device from the TrISH configuration file. For instance, the configuration function for the optical disk jukebox retrieves the path of the robotic changer and the mount point for the device. The configuration functions are optional.

The optional initialization and cleanup functions place the device in a known state and initialize access method block values. They are also responsible for retrieving dynamic configuration information for devices that are dynamically allocated.

The volume-handling functions are used to mount and unmount offline media. Some devices do not have removable media, so for these devices, the media handling access routines would not be defined.

The granule functions are used to open, close, allocate, and delete granules. These functions are required for all access methods. The open routine is responsible for positioning the media, starting any helper programs (like compress or uncompress), and opening the device using the `open()` system call. The close routine ends any helper programs and calls `close()` on the device and any other necessary function to prepare the device for the next open, mount, or unmount request. The allocate function is responsible for allocating space on a volume for

Table G.1. Access Method Routines

Access Method Routine	*Description*
	Configure functions
`dev_config`	Read special config info for a specific device
`devt_config`	Read special config info for the device type
	Initialize/Cleanup functions
`init`	Initialize device, state data structures and such
`dynam_init`	Further initialization of dynamically allocated devices
`cleanup`	Return device to know state, free state data structures
	Volume functions
`mount`	Mount a specific volume
`unmount`	Un-mount currently mounted volume
	Granule functions
`open`	Open a granule for reading or writing
`close`	Close a granule
`allocate`	Allocate space on a volume for a granule
`delete`	Delete a granule from a volume
	Block functions
`read`	Read a block from an open granule
`write`	Write a block to an open granule
`seek`	Seek to a specific block in an open granule

the granule. It will most likely need to consult and update its database for the current or next available volume. The delete function is responsible for freeing the space allocated by invalid granules. For devices like magnetic tape, another process may be required to consolidate good data onto one volume and free the volume to be used again.

The block functions are used to read, write, and seek the actual data on the device. The unit of transfer, the block size, is determined by the access routines and is defined to optimize the data transfer rate of the device. The read and write routines can implement optional processing of the data at the block level. For instance, the optical disk access methods implement CRC checking at the block level. At the simplest level, the read and write routines just call the operating system `read()` and `write()` system calls.

APPENDIX H

DATABASE ACCESS ROUTINES

There are database access functions for initializing and closing the database and adding, deleting, updating, querying, and sorting various database entries. These include granule entries, volume entries, releasable file entries, migratable file entries, and forced migration file entries. A list of the database access functions can be found in Table H.1.

Table H.1. Database Access Routines

Database Access Routine	*Description*
`dbinit()`	Open the TrISH database
`dbcleanup()`	Close the TrISH database
`dbinit_outdb()`	Open the outmigration database
`dbcleanup_outdb()`	Close the outmigration database
`add_granual()`	Add a granule entry
`delete_granual()`	Delete a granule entry
`obsolete_granual()`	Obsolete a granule
`update_granual_use()`	Update usage statistics for granule
`update_granual_flags()`	Update granule flags
`granuals_for_fmigid()`	Find all granules for a file
`granuals_for_fmigid_on_dev()`	Find granules for a file on a device
`obsolete_granual_list()`	Obsolete a list of granules
`next_obsolete_granual()`	Find the next obsolete granule
`add_inprocess()`	Add an inprocess entry
`delete_inprocess()`	Delete an inprocess entry
`update_inprocess()`	Update an inprocess entry
`next_inprocess()`	Find next inprocess entry
`total_inprocess_bytes()`	Calculate total of all inprocess entries
`add_migratable()`	Add a migratable entry
`readel_migratable()`	Find and delete next migratable entry
`add_releasable()`	Add a releasable entry
`update_releasable()`	Update releasable entry
`first_releasable()`	Find highest valued releasable entry
`total_releasable_bytes()`	Calculate total of all releasable entries
`add_forced()`	Add a file to forced migration list
`readel_forced()`	Find and delete next forced entry
`get_current_volume()`	Get current target volume
`get_next_available_volume()`	Get next free volume
`set_current_volume()`	Set the current target volume
`update_volume_use()`	Update volume use statistics
`update_volume_freespace()`	Update volume space statistics
`lock_volume_db_ent()`	Mark a volume as active
`unlock_volume_db_ent()`	Mark a volume as inactive
`unlock_all_volumes()`	Mark all volumes as inactive
`next_fmigid()`	Get next available fmigid value
`find_path_for_fmigid()`	Find file name associated with fmigid

APPENDIX I

MISCELLANEOUS TRISH PROGRAMS

A few additional commands and utilities have been provided by TrISH to enable the end user to set process flags and to help the system administrator manage the TrISH system. They are discussed in this section.

I.1 Setting File Migration Process Flags

As discussed in section 4.2.2, a process can customize the behavior of the file migration system by setting a number of process flags. The `trflags` command is used to set these process flags.

```
trflags {set|get} {retry|notransp|cancel} pid [on|off]
```

I.2 Filesystem Analysis

To help the system administrator determine if a filesystem is a good candidate for enabling file migration, the `trfsanal` program has been provided. This program gathers information about a filesystem and outputs a file that can be graphed. An example output file is shown in Figure I.1 and a graph of the data is shown in Figure I.2.

I.3 Retrieving Migration System Statistics

The file migration system routines in the kernel maintain a number of counters so that the activity of both migrated and nonmigrated files can be tracked. These routines count the number of `open()`, `close()`, and `read` requests, as well as a number of other requests, against both migrated and nonmigrated files. These routines also keep track of the delays incurred by processes that access migrated files.

```
#
# File system statistics
# Total number of directories scanned:            13049
# Total number of files scanned:                 192306
# Total Kilobytes in files scanned:             2950103K
#
# Average days since last access:                523 days
# Average days since last modification:         1532 days
# Average days not accessed since modified:     1008 days
#
# Files that are active:                           859 =  0%
# Files that are actively modified:                134 =  0%
# Files that have not been used since created:    6896 =  3%
#
#filename size.dat
#
#File Size
#
#              Number    % of    Cumm %    Space   % of    Cumm %
#  Size (K)   of files   files  of files   in K    Space  of Space
#-----------  --------   -----  --------  -------  -----  --------
         0      35014      18       18          0      0        0
         1      31649      16       34      31649      1        1
         2      22379      11       46      44758      1        2
         4      28764      14       61      98438      3        5
         8      26605      13       75     165504      5       11
        16      20968      10       85     248024      8       19
        32      14321       7       93     330084     11       31
        64       7490       3       97     333247     11       42
       128       2971       1       98     260403      8       51
       256       1014       0       99     177039      6       57
       512        652       0       99     260608      8       66
      1024        263       0       99     182177      6       72
      2048         93       0       99     136861      4       76
      4096         54       0       99     155378      5       82
      8192         52       0       99     267003      9       91
     16384         13       0       99     136195      4       95
     32768          3       0       99      74603      2       98
     65536          1       0      100      48132      1      100
    131072          0       0      100          0      0      100
    262144          0       0      100          0      0      100
    524288          0       0      100          0      0      100
   1048576          0       0      100          0      0      100
   2097152          0       0      100          0      0      100
2147483647          0       0      100          0      0      100
```

Figure I.1. Filesystem Analysis Graph

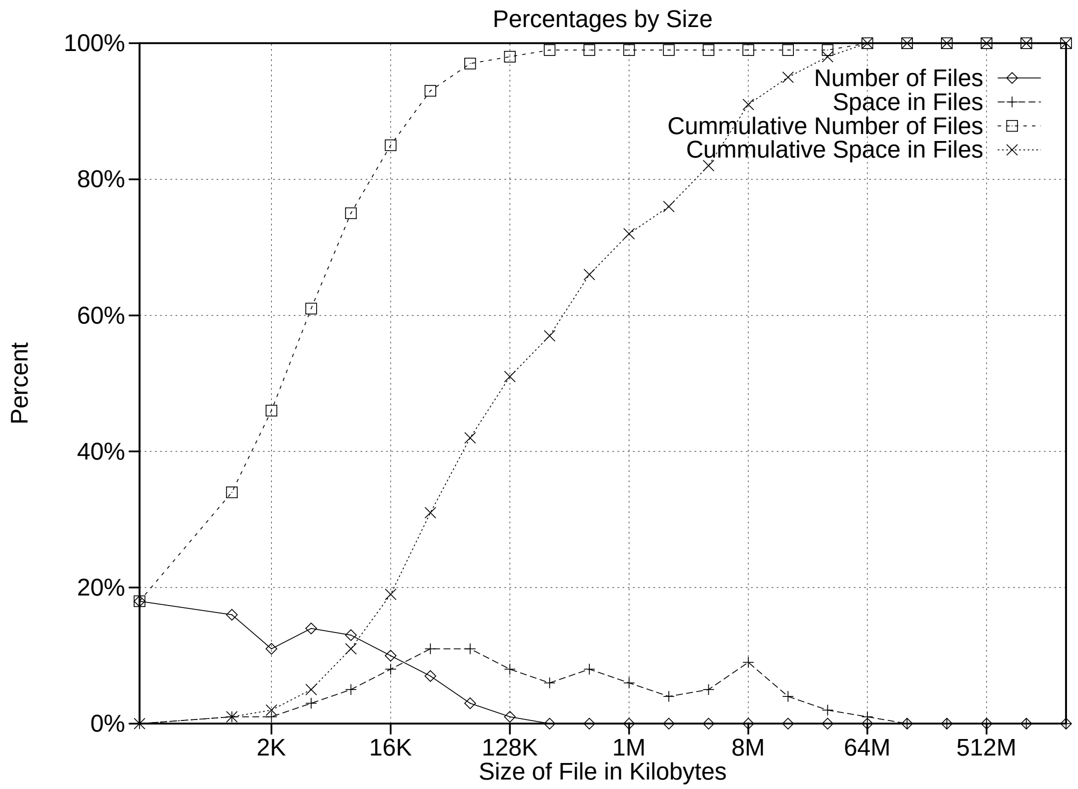

Figure I.2. Filesystem Analysis Graph

The `trkstat` command retrieves and displays the file migration kernel statistics. Below is an example of output from the `trkstat` command.

```
40 saaz> trkstat
 ----- Regular Stats ----- 1995-03-26 11:01:18.09
open = 254676
close = 254675
read = 1028941
read_odrl = 0
read_odnrl = 0
read_blk = 0
write = 664881
write_odrl = 0
write_odnrl = 0
write_blk = 0
trunc = 1552
unlink = 535
chown = 4
demigrated = 0
select = 0
lowspace = 0
nospace = 0
getattr = 978840
iinactive = 0
exec = 18
 ----- Migrated Stats ----- 1995-03-26 11:01:18.12
open = 6
close = 272
read = 41213
read_odrl = 5837
read_odnrl = 32761
read_blk = 2615
write = 0
write_odrl = 0
write_odnrl = 0
write_blk = 0
trunc = 0
unlink = 18
chown = 0
demigrated = 0
select = 0
lowspace = 0
nospace = 0
getattr = 202251
iinactive = 0
exec = 1
```

```
 ----- Delay Stats ----- 1995-03-26 11:01:18.13
.001 0 0
.002 0 0
.005 .014764 3
.01 .633248 83
.02 3.1633 218
.05 56.8865 1476
.1 44.8029 762
.2 4.71357 37
.5 7.20189 23
1 5.59684 8
2 4.59303 4
5 0 0
10 0 0
20 15.1961 1
50 0 0
100 0 0
200 0 0
500 0 0
1000 0 0
2000 0 0
5000 0 0
10000 0 0
20000 0 0
50000 0 0
```

APPENDIX J

THE TRCTL COMMAND

The TrISH control program, `trctl`, is used to control the TrISH migration daemon and its reload servers. Using it a system administrator can do the following:

- Start a reload server
- Stop a reload server
- Retrieve information about reload servers
- Shutdown the TrISH system
- Retrieve the status of `trishd`

Using the `trctl` program a user can do the following:

- Initiate batch priority reloads
- Cancel pending and in-process reload requests
- Check on the status of reload requests

The `trctl` command options are shown in Figure J.1.

```
USAGE: trctl cmd
    where cmd is one of:
            sh                   (shutdown)
            st                   (status)
            dt devname ...       (device start)
            dd devname ...       (device drain)
            ds [all|devname...] (device status)
            rl file ...          (reload file)
            ca file ...          (cancel file reload)
            cf fsid.fmigid ...  (cancel fmigid reload)
            rs                   (reload status)
```

Figure J.1. `trctl` Command Options

APPENDIX K

DEVICE MANAGEMENT

The UNIX system provides mechanisms for serializing access to some devices, like printers, but has no mechanism for serializing access to offline devices, like tape drives. Since an offline device may be shared between the TrISH system and, for example, the backup system, a serialization mechanism is needed.

K.1 The TrISH Device Manager

A standard part of the TrISH system is a device manager that guarantees serialized access to offline devices. When a process requires the use of an offline device, it sends a request to the device manager. If the device is available, it is allocated to the process. If it is already allocated to another process, the requesting process can choose to wait for the device to become available or it can try for another device.

The device manager provides generic device allocation. If a process needs an 8mm tape drive and there are four of them on the system, any one of the tape drives can be allocated to the requesting process. Only when all devices in a generic group are allocated does the request fail.

The grouping of generic devices is also supported. For instance, a system has four optical disk drives in two different jukebox. A process requires access to an optical disk in the storage slot of jukebox A. Since the robot mechanism only has access to the optical disks in its own storage slots, the requesting process must be able to specify that it needs an optical disk drive in jukebox A.

K.2 Allocating and Releasing A Device

The device management system has both command line and C library access mechanisms. The command line interface consists of the `devalloc` and `devrelease` commands. These commands can be easily integrated into UNIX shell scripts, since they print shell commands to set environment variables. The output of `devalloc` can be given to the `eval` command to set these environment variables, which are then used in the shell script. A simple example is shown below.

```
101 saaz> set devcmds=`devalloc hpmo_box1`
102 saaz> echo $devcmds
setenv DEV_NAME hpmo_drive1;
setenv DEV_TYPE hpmoac;
setenv DEV_OPER_NAME hpmo_drive1;
setenv DEV_PATH /dev/sd1c;
103 saaz> eval $devcmds
104 saaz> tar -cvf $DEV_PATH
105 saaz> devrelease $DEV_NAME
```

The C library interface consists of two routines that are analogous to the command line routines. They are the `devmgr_devalloc()` and `devmgr_devrelease()` routines. They are used in a similar manner to the command line interface programs.

APPENDIX L

COMMUNICATING WITH THE OPERATOR

On some occasions, a daemon process may need to send a message to and get a reply from the computer operator or the system administrator. For instance, if a reload server needs a tape loaded into a tape drive, there should be a simple way it can correspond with the operator. The operator should be able to cancel the request without killing the daemon if, for instance, the tape is lost or damaged.

On MVS/ESA, IBM's mainframe operating system, a process can send and receive messages from the operator via a standard mechanism. There is no such mechanism in UNIX.

L.1 The `oprq` Command

The operator question-and-answer utility (`oprq`) has been provided with TrISH to fill the requirement. It has a number of ways to communicate with the operator, such as sending mail, writing messages to a tty port like the system console, and opening an X-Window on any X-server. It can be configured to try a number of communication paths all at once, and if an answer is not received in a specified amount of time, it will try a different set of communication paths.

Below is an example `oprq` command. The X-window message it displayed is shown in Figure L.1.

```
oprq -yn -y Continue -n Cancel "Please mount tape 1234 in drive ABC"
```

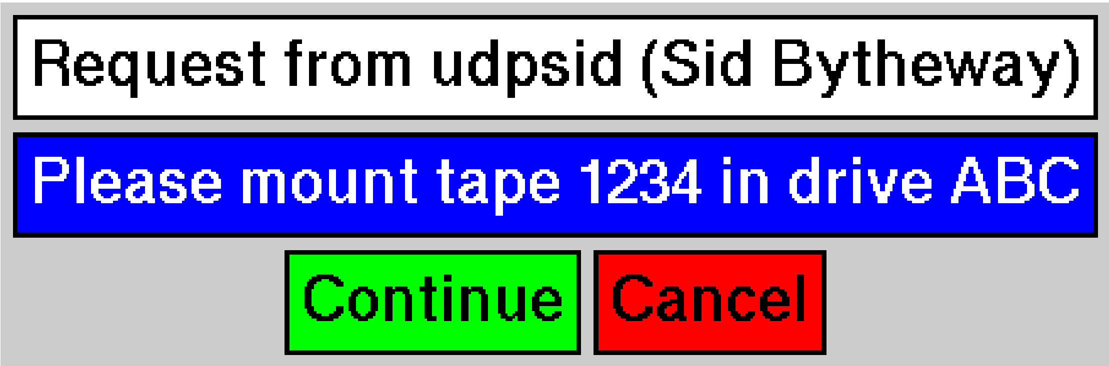

Figure L.1. Sample X-window from the `oprq` Command

REFERENCES

[1] Ann Louise Chervenak. *Tertiary Storage: An Evaluation of New Applications.* PhD thesis, University of California at Berkeley, 1994.

[2] David A. Patterson. Terabytes >> Teraflops (Or Why Work on Processors When I/O is Where the Action Is? Abstract from Keynote address at the ACM SIGMETRICS Conference in Santa Clara, CA on May 13, 1993., May 1993.

[3] General Atomics. *The UniTree Virtual Disk System — An Overview.*

[4] Jonathan Hahn, Bob Henderson, Ruth Iverson, Shri Lohia, Alan Poston, Tom Proett, Bill Ross, Mark Tangney, and Dave Tweten. NAStore External Reference Specification. NAStore Specification Document, September 1991.

[5] Robert L. Henderson and Alan Poston. MSS-II and RASH, A Mainframe UNIX Based Mass Storage System with a Rapid Access Storage Hierarchy File Management System. In *Proceedings of the Winter 1989 USENIX Conference.* The USENIX Association, 1989.

[6] International Business Machines Corporation. *Data Facility Hierarchical Storage Manager: Version 2, Release 4.0, General Information.* Publication Number GH35-0092-4.

[7] International Business Machines Corporation. *Licensed Program Specifications for Data Facility Hierarchical Storage Manager: Version 2, Release 5, Modification Level 0.* Publication Number GH35-0096-06.

[8] Irlam, Usenet community trust. Unix File Size Survey. Internet Unix File Size Survey, October 1993.

[9] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley Publishing Company, November 1989.

[10] Jane Majkiewicz. Building a Storage Strategy. *RS/Magazine*, pages 38–46, April 1994.

[11] Ethan L. Miller and Randy H. Katz. An Analysis of File Migration in a Unix Supercomputing Environment. In *USENIX Technical Conference Proceedings - Winter 1993*, pages 421–433. University of California, Berkeley, The USENIX Association, 1993.

[12] Michael John Muuss, Terry Slattery, and Donald F. Merritt. BUMP, The BRL/USNA Migration Project. Description Included in BUMP Distribution.