# HighLight: Using a Log-structured File System for Tertiary Storage Management

by John T. Kohl

## Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

**Committee:**

Professor M.A. Stonebraker
Research Advisor

(Date)

* * * * * * *

Professor J.K. Ousterhout
Second Reader

(Date)

## Abstract

Log-structured file systems (LFSs) were developed to eliminate latencies involved in accessing disk devices, but their sequential write patterns also match well with tertiary storage characteristics. Unfortunately, existing versions only manage memory caches and disks, and do not support a broader storage hierarchy. Robotic storage devices offer huge storage capacity at a low cost per byte, but with large access times. Integrating these devices into the storage hierarchy presents a challenge to file system designers.

HighLight extends 4.4BSD LFS to incorporate both secondary storage devices (disks) and tertiary storage devices (such as robotic tape jukeboxes), providing a hierarchy within the file system that does not require any application support. This report presents the design of HighLight, proposes various policies for automatic migration of file data between the hierarchy levels, and presents initial migration mechanism performance figures.

Log-structured file systems, with their sequential write patterns and large transfer sizes, extend naturally to encompass a storage hierarchy. HighLight is flexible enough to support a variety of potential migration policies. When data are disk-resident, access is nearly as fast as access to files in the base 4.4BSD LFS. When disk arm contention is absent, transfers to tertiary devices can run at nearly the tertiary device transfer speed. [1]

[1]This report is substantially derived from an article originally published in the USENIX Association Conference Proceedings, January 1993. Permission has been granted by the USENIX Association to incorporate portions of that paper in other publications. Portions of this report Copyright ©USENIX Association, 1993.

This research was sponsored in part by the University of California and Digital Equipment Corporation under Digital's flagship research project "Sequoia 2000: Large Capacity Object Servers to Support Global Change Research." Other industrial and government partners include the California Department of Water Resources, United States Geological Survey, MCI, ESL, Hewlett Packard, RSI, SAIC, PictureTel, Metrum Information Storage, and Hughes Aircraft Corporation. This work was also supported in part by Digital Equipment Corporation's Graduate Engineering Education Program.

## 1 Introduction

HighLight combines both conventional disk secondary storage and robotic tertiary storage into a single file system. It builds upon the 4.4BSD Log-structured File System (LFS) [11], which derives directly from the Sprite LFS [9], developed at the University of California at Berkeley by Mendel Rosenblum and John Ousterhout as part of the Sprite operating system. LFS is optimized for writing data, whereas most file systems (e.g. the BSD Fast File System [4]) are optimized for reading data.

Since log-structured file systems are optimized for write performance, they are a good match for the write-dominated environment of archival storage. However, system performance will depend on optimizing read performance, since LFS already optimizes write performance. Therefore, migration policies and mechanisms should arrange the data on tertiary storage to improve read performance.

The development of HighLight was predicated on the premise that the LFS on-disk data layout, with large sequential writes, is a natural fit for typical tertiary storage devices (such as robotic tape libraries). HighLight uses the same data format on both secondary and tertiary storage, transferring entire LFS *segments* (portions of the log) between the levels of the storage hierarchy. It arranges file data in these segments to improve read performance.

HighLight currently collects to-be-migrated files into to-be-migrated segments, implementing a file-oriented migration policy. It is currently running in the laboratory with an automatic migration mechanism as well as a manual migration tool. HighLight can migrate files to tertiary storage and later demand-fetch them back onto a disk cache so that applications can access them. Applications never need know (except by reduced access times) that files are not always resident on secondary storage.

The remainder of this report presents HighLight's mechanisms, gives some preliminary performance measurements, and speculates on some useful migration policies. HighLight's context within Sequoia is described in Section 2. Section 3 provides a sketch of the basic log-structured file system, which is followed in Section 4 by a discussion of the storage and migration model. Section 5 briefly discusses potential migration policies; Section 6 describes HighLight's architecture. Some preliminary measurements of the system performance appear in Section 7. A comparison of HighLight to existing related work in policy and mechanism design is in Section 8. The report concludes with a summary and directions for future work.

## 2 Sequoia Background

HighLight was developed to provide a data storage file system for use by Sequoia researchers. Project Sequoia 2000 [16] is a collaborative project between computer scientists and earth science researchers to develop the necessary support structure to enable global change research on a larger scale than current systems can support. The bulk of the on-line storage for Sequoia will be provided by a 600-cartridge Metrum robotic tape unit; each cartridge has a capacity of 14.5 gigabytes for a total of nearly 9 terabytes. Sequoia also will use a collection of smaller robotic tertiary devices, including a Hewlett-Packard 6300 magneto-optic changer with total capacity of about 100GB and a Sony write-once optical disk jukebox with total capacity of about 327GB.

The variety of robotic storage devices available for Sequoia's use resulted in the definition of an abstract robotic device interface, called *Footprint*. HighLight will have exclusive rights to some portion of the tertiary storage space, accessing it through Footprint. While it knows the capacities of the tertiary volumes (be they tapes or optical disks), the abstract interface unburdens HighLight from needing to understand the details of a particular device.

HighLight is one of several file management avenues under exploration as a supporting storage technology for this research. Sequoia needs some storage management system to manage the storage hierarchy so that it is not intrusive to the global change researchers' work. Alternative storage management efforts under investigation include the Inversion file system support in the POSTGRES database system [7] and the Jaquith manual archive system [6] (which was developed for other uses, but is under consideration for Sequoia's use). The systems are not mutually exclusive; the best solution for Sequoia may involve some combination of these elements (perhaps Inversion and/or POSTGRES will be hosted on top of HighLight). When each system is in a suitable condition, there will be a "bake-off" to compare and contrast the systems and see how well they support an actual work load.

## 3 LFS Primer

The primary characteristic of LFS is that all data are stored in a segmented log. The storage consists of large contiguous spaces called *segments* that may be threaded together to form a linear log. LFS divides the disk into 512KB or 1MB segments and writes data sequentially within each segment. New data are appended to the log, and periodically the system checkpoints its file system metadata state. During recovery the system will roll-forward from the last checkpoint, using the information in the log to recover the state of the file system at failure. Obviously, as data are deleted or replaced, the log contains blocks of invalid or obsolete data, and the system must coalesce this wasted space to generate new, empty segments for the log. Disk space is reclaimed by copying valid data from dirty segments to the tail of the log and marking the emptied segments as clean.

4.4BSD LFS shares much of its implementation with the Berkeley Fast File System (FFS) [4]. It has two auxiliary data structures not found in FFS: the *segment summary table* and the *inode map*. The segment summary table contains information describing the state of each segment in the file system. Some of this information, such as indications of whether the segment is clean or dirty, is necessary for correct operation of the file system, while other information is used to improve the performance of the cleaner, such as counts of the number of live data bytes (data that are still accessible to a user, i.e. not yet deleted or replaced) in the segment. The inode map contains the current disk address of each file's inode, as well as some auxiliary information used for file system bookkeeping. In 4.4BSD LFS, both the inode map and the segment summary table are contained in a regular file, called the *ifile*.

When reading files, the only difference between LFS and FFS is that the inode's location is variable. Once the system has found the inode (by indexing the inode map), LFS reads occur in the same fashion as FFS reads, by following direct and indirect block pointers[2].

When writing, LFS and FFS differ substantially. In FFS, each logical block within a file is assigned a location upon allocation, and each subsequent operation (read or write) is directed to that location. In LFS, data are written to the tail of the log each time they are modified, so their location changes. This requires that their index structures (indirect blocks, inodes, inode map entries, etc.) be updated to reflect their new location, and these index structures are also appended to the log.

Each segment of the log may contain several *partial segments*. A partial segment is considered an atomic update to the log, and is headed by a segment summary cataloging its contents. A sample summary is shown in Table 1. The summary also includes a checksum to verify that the entire partial segment is intact on disk and provide an assurance of atomicity.

In order to provide the system with a ready supply of empty segments for the log, a user-level process called the *cleaner* garbage collects free space from dirty segments. The cleaner selects one or more dirty segments to be cleaned, appends all valid data from those segments to the tail of the log, and then marks those segments clean. The cleaner communicates with the file system by reading the ifile and calling a handful of LFS-specific system calls. The cleaner being a user-level process simplifies the adjustment of cleaning policies.

For recovery purposes the file system takes periodic checkpoints. During a checkpoint the address of the most recent *ifile* inode is stored in the superblock so that the recovery agent may find it. During recovery the threaded log is used to roll forward from the last checkpoint. During recovery, the system scans the log, examining each partial segment in sequence. When an incomplete partial

[2] In fact, LFS and FFS share this indirection code in 4.4BSD.

| Field name | Bytes | description |
|---|---|---|
| ss_sumsum | 4 | check sum of summary block |
| ss_datasum | 4 | check sum of data |
| ss_next | 4 | disk address of next segment in log |
| ss_create | 4 | creation time stamp |
| ss_nfinfo | 2 | number of file info structures |
| ss_ninos | 2 | number of inodes in summary |
| ss_flags | 2 | flags; used for directory operations |
| ss_pad | 2 | extra space for word-alignment |
| . . . | 12 per distinct file in partial segment + 4 per file block | file block description information |
| . . . | 4 per inode block | inode block disk addresses |

**Table 1**: A partial segment summary block. The end of the summary contains file block information, describing the file data blocks in the partial segment, and inode disk addresses, locating inode blocks within the partial segment.

segment is found, recovery is complete and the state of the filesystem is the state as of the last complete partial segment.

Figure 1 illustrates the on-disk data structures of 4.4BSD LFS. The on-disk data space is divided into segments. Each segment has a summary of its state (whether it is **c**lean, **d**irty, or **a**ctive). Dirty segments contain live data. At the start of each segment there is a summary block describing the data contained within the segment and pointing to the next segment in the threaded log. In Figure 1 there are three segments, numbered 0, 1, and 2. Segment 0 contains the current tail of the log. New data are being written to this segment, so it is active. It is also dirty since it contains data that are still valid. Once Segment 0 fills up, it will cease being active and the system will activate and begin writing to Segment 1, which is currently clean and empty. Segment 2 was written just before Segment 0; it is dirty.

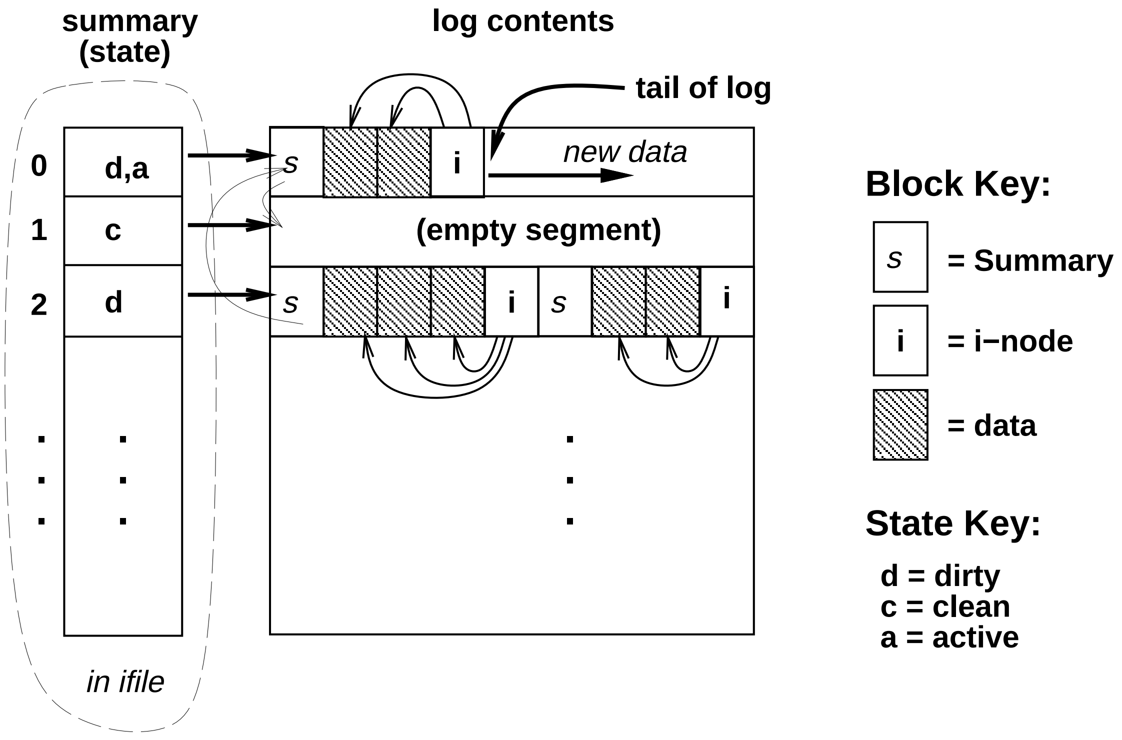

**Figure 1**: LFS data layout.

## 4 Storage and Migration model

HighLight extends LFS, using a “disk farm” to provide rapid access to file data, and one or more tertiary storage devices to provide vast storage. It manages the storage and the migration between the two levels. The basic storage and migration model is illustrated in Figure 2. Application programs see only a “normal” filesystem, accessible through the usual operating system calls. They may notice a degradation in access time due to the underlying hierarchy management, but they need not take any special actions to utilize HighLight.

HighLight has a great deal of flexibility, allowing arbitrary data blocks, directories, indirect blocks, and inodes to migrate to tertiary storage at any time. It uses the basic LFS layout to manage the on-disk storage and applies a variant on the cleaning mechanism to provide the migration

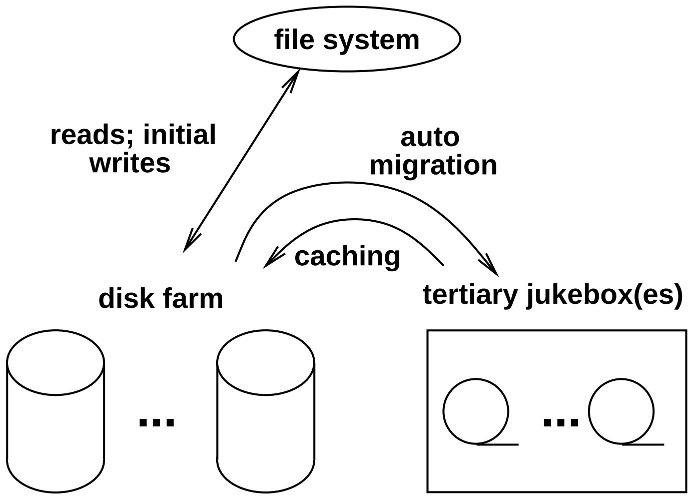

**Figure 2**: The storage hierarchy. A collection of disks provides initial storage for new files, and fast access to tertiary-resident data. An automatic migration process copies file data to LFS segments on tertiary storage when appropriate. If migrated file data are needed, the containing segment(s) are cached on disk and the file data are read from the cached copy.

mechanism. A natural consequence of this layout is the use of LFS segments for the tertiary-resident data representation and transfer unit. By migrating segments, it is possible to migrate some data blocks of a file while allowing others to remain on disk if a file's blocks span more than one segment.

Data begin life on the "disk farm" when they are created. A file (or part of it) may eventually migrate to tertiary storage according to a migration policy. The to-be-migrated data are moved to an LFS segment in a staging area, using a mechanism much like the cleaner's normal segment reclamation. When the staging segment is filled, it is written to tertiary storage as a unit and a new staging segment is created.

When tertiary-resident data are referenced, their containing segment(s) are fetched into the disk cache. These read-only cached segments share the disk with active regular segments. Figure 3 shows a sample tertiary-resident segment cached in a disk segment. Data in cached tertiary-resident segments are not modified in place on disk; rather, any changes are appended to the LFS log in the normal fashion. Since read-only cached segments never contain the sole copy of a block (there is a copy on tertiary storage as well), they may be discarded from the cache at any time if the space is needed for other cache segments or for new data.

The key combination of features that HighLight provides are: the ability to migrate all file system data (not just file contents, but also directories, inodes, and indirect blocks); making tertiary placement decisions made at migration time, instead of file creation time; migrating data in units of LFS segments; migrating using user-level processes; and implementing migration policy with a user-level process.

## 5 Migration Policies

Because HighLight includes a storage hierarchy, it must move data up and down the hierarchy. Migration between hierarchy levels can be targeted at several objectives, not all of which may be simultaneously achievable:

- Minimize file access times (to applications)
- Maximize file system data throughput (to applications)
- Minimize bookkeeping overhead
- Maximize disk utilization
- Minimize required ratio of secondary to tertiary storage capacity

Migration policies may make different tradeoffs between these goals. Locality of reference is likely to be present in most application loads, and so is likely to be assumed by migration policies.

Before describing the proposed migration policies, it is important to state the initial assumptions regarding file access patterns, which are based on previous analyses of systems [5, 18, 13]. The basic assumptions are that file access patterns are skewed, such that most archived data are never re-read. However, some archived data will be accessed, and once archived data became active again, they will be accessed many times before becoming inactive again.

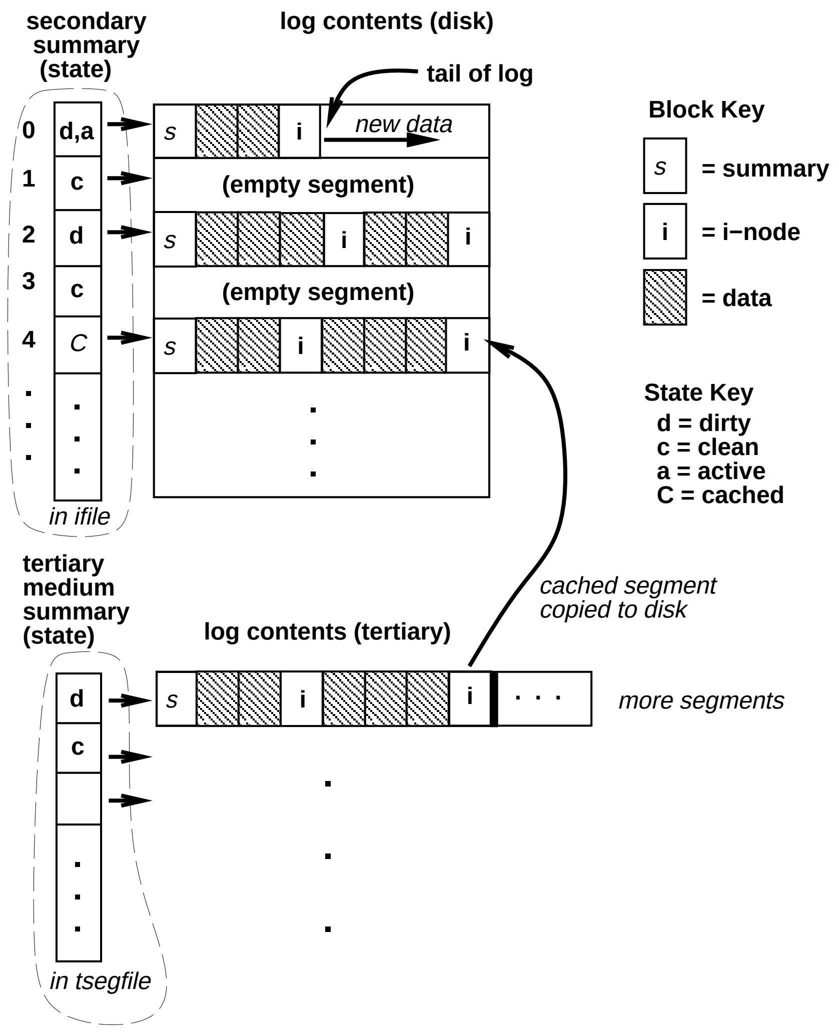

**Figure 3**: HighLight's data layout. Tertiary-storage resident segments are stored in the same format as secondary-storage segments; demand-fetching copies tertiary segments to secondary storage. A migrator process ejects cached segments when they lay dormant.

Since HighLight optimizes writes by virtue of its logging mechanism, migration policies must be aimed at improving read performance. When data resident on tertiary storage is cached on secondary storage and read, the migration policy should have optimized the layout so that these read operations are as inexpensive as possible. There needs to be a tight coupling between the cache fetch and migration policies.

HighLight has one primary tool for optimizing tertiary read performance: the segment size. When data are read from tertiary storage, a whole 1MB segment (which is the equivalent of a cache line in processor caches) is fetched and placed in the segment cache, so that additional accesses to data within the segment proceed at disk access rates. Policies used with HighLight should endeavor to cluster "related" data in a segment to improve read performance. The determination of whether data are "related" depends on the particular policy in use. If related data will not fit in the one segment, then their layout on tertiary storage should be arranged to admit a simple prefetch mechanism to reduce latencies for reads of the "related" data in the clustered segments.

Given perfect predictions, policies should migrate data that provides the best benefit to performance (which could mean something like migrating files that will never again be referenced, or referenced after all other files in the cache). Without perfect knowledge, however, migration policies need to estimate the benefits of migrating a file or set of files. Some estimating policies that will be evaluated with HighLight are presented below. All the possible policy components discussed below require some additional mechanism support beyond that provided by the basic 4.4BSD LFS. They require some basic migration bookkeeping and data transfer mechanisms, which are described in the Section 6.

The following sections examine some possible migration policies in two parts: writing to tertiary storage and caching from tertiary storage. Note that a running system is not restricted to a single policy; some of the policies can coexist without undue interference.

## 5.1 Access time-based rankings

An access time-based policy selects files for migration based on the interval since their last use, preferentially retaining active files on disk. This policy is based on locality of temporal reference: active files are likely to be accessed soon, while inactive files are not.

Earlier studies by Lawrie et. al. [3] and Smith [14] conclude that time-since-last-access alone does not work well for selecting files as migration candidates; they recommend using a weighted space-time product (STP) ranking metric, taking the time since last access, raised to a small power (possibly 1), times file size raised to a small power (possibly 1). Strange [18] evaluated different variations on the STP scheme for a typical networked workstation configuration. Those evaluations considered different environments, but generally agreed on the space-time product as a good metric. Whether these results still work well in the Sequoia environment is something HighLight will evaluate.

The space-time product metric has only modest requirements on the mechanisms, needing only the file attributes (available from the base LFS) and a whole-file migration mechanism. That mechanism is described below in Section 6. The current migrator in fact uses STP with exponents of 1 for the file size and access times.

## 5.2 Choosing block ranges

In the simplest policies, HighLight could use whole-file migration, with mechanism support based on file access and modification times contained in the inode. However, in some environments whole file migration may be inadequate. In UNIX-like distributed file system environments, most files are accessed sequentially and many of those are read completely [1]. Scientific application checkpoints, such as those generated by some of the earth science researchers' analysis or simulation programs, tend to be read completely and sequentially. (Such checkpoints typically dump the internal state of a computation to files, so that the state may be reconstituted and the computation resumed at a later time. When they resume, they read in the entire state from the checkpoint file.) In these cases, whole file migration makes sense.

However, database files tend to be large, may be accessed randomly and incompletely (depending on the application's queries), and in some systems [15] are never overwritten. Even simple hash-based database packages such as the Berkeley hash package [12] can yield non-sequential access patterns, depending on the application. Block-based migration can be useful, since it allows old, unreferenced data within a file to migrate to tertiary storage while active data in the same file remain on secondary storage.

In order to provide migration on a finer grain than whole files, HighLight must keep some information on each disk-resident data block in order to assist the migration decisions. Keeping information for each block on disk would be exorbitantly expensive in terms of space, and often unnecessary. A compromise solution is to keep track of access ranges within a file, with the potential to resolve down to block granularity. In this way files that are accessed sequentially and completely have only a single record, while database files might potentially have a separate record for each data block on disk. Each record's chunk of blocks may then be separately considered for migration.

There is a tradeoff inherent in this middle ground: by storing more information (at a fine sub-file granularity), the policy can make better decisions at the cost of requiring more bookkeeping space; less information (coarser granularity) may result in worse decisions (perhaps causing a migration in or out when some of the blocks in the range should remain where they are) but consumes less overhead. The dynamic nature of the granularity attempts to get the most benefit for the least overhead.

Such tracking requires a fair amount of support from the mechanism: access to the sequential block-range information, which implies mechanism-supplied and updated records of file access sequentiality. There is no clear implementation strategy for this policy yet.

## 5.3 Namespace Locality

When dealing with a collection of small files, it is more efficient to migrate several related files at once. A file namespace can identify these collections of “related” files (*units*); such directory trees or sub-trees can be migrated to tertiary storage together. This is useful primarily in an environment where whole subtrees are related and accessed at nearly the same time, such as software development environments.

The file size/access time criteria discussed above can be applied to these units. The space-time metric then becomes a “unitsize”-time product, where unitsize is the aggregate size of all the component files, and time-since-last-access is the minimum over the files considered. If a unit is too large for a single tertiary segment, a natural prefetch policy on a cache miss is to load the missed segment and prefetch remaining segments of the unit. Migrated units should then be clustered according to their position in the naming hierarchy, so that units near each other in the naming tree are near each other on tertiary storage.

This policy could cause problems if there are large units with only a handful of active files. In these cases, the unit would never be migrated to tertiary storage, even though the inactive files are polluting the active disk area. To alleviate this, the policy can establish secondary criteria for units, such as ignoring access times on the most-recently-accessed file if it has not been modified recently. This enables migration of units containing mostly-dormant files. Any active files in the unit would end up in a tertiary segment cached on disk, which segment would be subject to ejection from the cache under the normal selection criteria. If an active file is stable (not being modified), say if it is just a popular satellite image file, this migration will not consume any more tertiary storage space than would be consumed by delaying migration until the file is truly dormant. It may, however, consume more disk space if its active portions do not completely fill their segments in the cache.

Units with unstable (changing) files should probably not be migrated unless the locality is not very important, since the unstable files will soon be written to a new disk segment. If the file later stabilizes and is migrated to a tertiary segment, the unit may end up scattered among several tertiary segments (potentially on different volumes). Rearranging tertiary-resident data (as described below) may assist in remedying this pitfall.

The primary additional requirement of a namespace clustering policy is a way to examine file system trees without disturbing the access times; this is possible to do with a user program which just walks the file tree examining file access times, since BSD filesystems do not update directory access times on normal directory accesses, and the file access time may be examined without modification.

## 5.4 Caching tertiary-stored data

Besides considering how to choose file data for migration, HighLight must consider how to manage its secondary storage cache of tertiary storage segments. The cache is mostly read-only; the exception is segments being assembled before transfer to tertiary storage.

The cache stages whole segments at a time from tertiary storage. In addition, the cache may prefetch segments it expects to be needed in the near future. These prefetching decisions may be based on hints left by the migrator when it wrote the data to tertiary storage, or they may be based on observations of recent accesses. Cache flushing could be handled by any of the standard policies: LRU, random, working-set observations, etc.

The cache provides opportunities to improve the tertiary access and overall file system performance by considering access patterns and perhaps copying file data into new tertiary segments (in a process similar to the initial copies for migration). By rearranging file data, the policy may improve access to related data by clustering them on a single volume. By considering overall file system access patterns, it may improve performance by avoiding disk contention when writing fresh tertiary segments to tertiary storage.

**Rearranging tertiary segments**

Data access patterns to tertiary-backed storage may change over time (for example, if several satellite data sets are loaded independently, and then those data sets are analyzed together). Performance may be boosted in such cases by reorganizing the data layout on tertiary storage to reflect the most prevalent access pattern(s). This reorganization can be accomplished by re-writing and clustering cached segments to a new storage location on the tertiary device when segment(s) are ejected from the cache. One good reason to do so would be to move segments to a different tertiary volume with access characteristics more suited to those segments.

This approach could be a problem if the extra writes from the rearrangements begin interfering with demand-fetch read traffic to the tertiary devices. This policy also unfortunately tends to increase the consumption of tertiary storage, and it is not clear how closely segment cache flushes are tied to access locality. A better approach might be to rewrite segments to tertiary storage as they are read into the cache. This is more likely to reflect true access locality.

Implicit in this scheme is the need to choose which cached segments should be rewritten to a new location on tertiary storage. All of the questions appropriate to migrating data in the first place are appropriate, so the overhead involved here might be significant (and might be an impediment if cache flushes are demand-driven and need to be fast reclaims).

A variant on this scheme is to maintain several segment replicas on tertiary storage, and to have the staging code simply read the o“closest” copy, where close means quickest access—whether that means seeking on a volume already in a drive, or selecting a volume that will incur a shorter seek time to the proper segment, or perhaps a volume with all the other segments that should be prefetched with the demand-fetched segment. One potential problem with this approach is the bookkeeping associated with determining when a tertiary-resident segment contains valid data (in order to allow reclamation of empty tertiary media). This problem could be sidestepped simply by not counting the replicas as live data.

This policy will require additional identifying information on each cache segment to indicate an appropriate locality of reference patterns between segments. Such information could be a segment fetch timestamp or the user-id or process-id responsible for a fetch, and could be maintained by the process servicing demand fetch requests and shared with the migrator. The variant would require a catalog or other index to find the replicas and a method to choose the "closest" one.

**Writing fresh tertiary segments**

HighLight needs to schedule the writing of migrated data to its tertiary device. Once a tertiary segment is assembled in the staging segment, it needs to be written to the tertiary volume. This write could be performed immediately upon completion of the assembly, or it could be delayed. As is seen below, performance may suffer (due to disk arm contention) if the new tertiary segments are copied to tertiary storage at the same time as other data are staged to another fresh cache-resident tertiary segment. This would occur if completed segments were copied out immediately after preparation and a process was migrating several segments' worth of file data. This suggests delaying segment writes to a later idle period when there will be no contention for the disk drive arm. Of course, if no such idle period arises, then this policy consumes some extra reserved disk space (the uncopied segments cannot be reclaimed until they are copied out tertiary storage ) and essentially reverts to the original style of copying to tertiary storage immediately upon completing the tertiary segment assembly (but with a several-segment deep pipeline between completion and copying).

This modification does not require extra mechanisms than those provided by the basic cache control: an indication of whether the segment is clean (copied to tertiary storage) and whether its migration or ejection is in progress.

### 5.5 Supporting migration policies

To summarize, there is potential use uses for (at least) the following mechanism features in an implementation:

- Basic migration bookkeeping (cache lookup control, data movement, etc.)
- Whole-file migration
- Directories and file system metadata migratable
- Grouping of files by some criterion (namespace)
- Cache fill timestamps/uid/pid
- Sequential block-range data (per-file)

The next section presents the current design and implementation of HighLight, which covers the first three features. File grouping could be added with a new user-level migrator process; cache fill annotations and sequential block-range data would require additional in-kernel file system support.

## 6 HighLight Design and Implementation

In order to provide "on-line" access to a large data storage capacity, HighLight manages secondary and tertiary storage within the framework of a unified file system based on the 4.4BSD LFS. The discussion here covers HighLight's basic components, block addressing scheme, secondary and tertiary storage organizations, migration mechanism, and other implementation details.

### 6.1 Components

HighLight extends 4.4BSD LFS by adding several new software components:

- A second cleaner, called the *migrator*, that collects data for migration from secondary to tertiary storage
- A disk-resident segment cache to hold read-only copies of tertiary-resident segments
- A kernel space/user space interface module which requests I/O from the user-level processes, implemented as a pseudo-disk driver
- A pseudo-disk driver that stripes multiple devices into a single logical drive
- A pair of user-level processes (the service process and the I/O process) to access the tertiary storage devices on behalf of the kernel.

Besides adding these new components, HighLight slightly modifies various portions of the user-level and kernel-level 4.4BSD LFS implementation (such as changing the minimum allocatable block size, adding conditional code based on whether segments are secondary or tertiary storage resident, etc.).

## 6.2 Basic operation

HighLight implements the normal filesystem operations expected by the 4.4BSD file system switch. When a file is accessed, HighLight fetches the necessary metadata and file data based on the traditional FFS inode's direct and indirect block pointers. (The pointers are 32 bits, but they address 4-kilobyte units, so a single filesystem and a single file is limited to 16 terabytes. There is no support for block fragments in either the base 4.4BSD LFS or in HighLight.) The block address space appears uniform, so that HighLight just passes the block number to its I/O device driver. The device driver maps the block number to whichever physical device stores the block (a disk, an on-disk cached copy of the block, or a tertiary volume).

The migrator process periodically examines the collection of on-disk file blocks, and decides (based upon some policy) which file data blocks and/or metadata blocks should be migrated to a tertiary volume. Those blocks are then assembled in a "staging segment" addressed by the block numbers the segment will use on the tertiary volume. The staging segment is assembled on-disk in a dirty cache line, using the same mechanism used by the cleaner to copy live data from an old segment to the current active segment. When the staging segment is filled, the kernel-resident part of the file system requests the server process to copy the dirty line (the entire 1MB segment) to tertiary storage. The request is serviced asynchronously, so that the migration control policies may choose to move multiple segments in a single logical operation for transfer efficiency.

The migrator is a separate process from the 4.4BSD LFS cleaner. Although it uses very similar mechanisms to assemble data into staging segments, its selection policies and function are sufficiently different that integrating it with the cleaner does not make much sense. Keeping them separate also allows migration and cleaning to proceed simultaneously.

Disk segments can be used to cache tertiary segments. Since the cached segments are almost always read-only copies of the tertiary-resident version, cache management is relatively simple, because read-only lines may be discarded at any time. Caching segments sometimes contain freshly-assembled tertiary segments; they are quickly scheduled for copying out to tertiary storage.

As in the normal LFS, when file data are updated, a new copy of the changed data is appended to the current on-disk log segment; the old copy remains undisturbed until its segment is cleaned or ejected from the cache. HighLight doesn't clean cached segments on disk; any cleaning of tertiary-resident segments would be done directly with the tertiary-resident copy in a process similar to the regular on-disk cleaning process. However, HighLight does not currently implement a tertiary volume cleaner.

If a process requests I/O on a file for which some necessary metadata or file data are stored on tertiary storage, a cached segment may satisfy the request. If the segment containing the required data is not in the cache, the kernel requests a demand fetch from the service process and waits for a reply. The service process finds a reusable segment on disk and directs the I/O process to fetch the necessary tertiary-resident segment into that segment. When that is complete, the service process registers the new cache line in the cache directory and calls the kernel to restart the file I/O.

The service or I/O process may choose unilaterally to eject or insert new segments into the cache. This allows them to prefetch multiple segments, perhaps based on some policy, hints, or historical access patterns.

## 6.3 Addressing tertiary storage

HighLight uses a uniform block address space for all devices in the filesystem. A single HighLight filesystem may span multiple disk and tertiary storage devices. Figure 4 illustrates the mapping of block numbers onto disk (secondary) and tertiary devices. Block addresses can be considered as a pair: (segment number, offset). The segment number determines both the volume (disk device, tape cartridge, or jukebox platter) and the segment's location within the volume. The offset identifies a particular block within the segment.

HighLight allocates a fixed number of segments to each tertiary volume. Since some media may hold a variable amount of data (e.g. due to device-level compression), this number is set to be the maximum number of segments the volume is expected to hold. If the compression factor exceeds the expectations, however, all the segments will fit on the volume and some storage at its end may be wasted. HighLight can tolerate less-efficient-than-expected compression on tertiary storage since it can keep writing segments to a volume until the drive returns an "end-of-medium" indication, at which point the volume is marked full and the last (partially written) segment is re-written onto the next volume.

When HighLight's I/O driver receives a block address, it simply compares the address with a table of component sizes and dispatches to the underlying device holding the desired block. Disks are assigned to the bottom of the address space (starting at block number zero), while tertiary storage is assigned to the top (starting at the largest block number). Tertiary media are still addressed with increasing block numbers, however, so that the end of the first volume is at the largest block

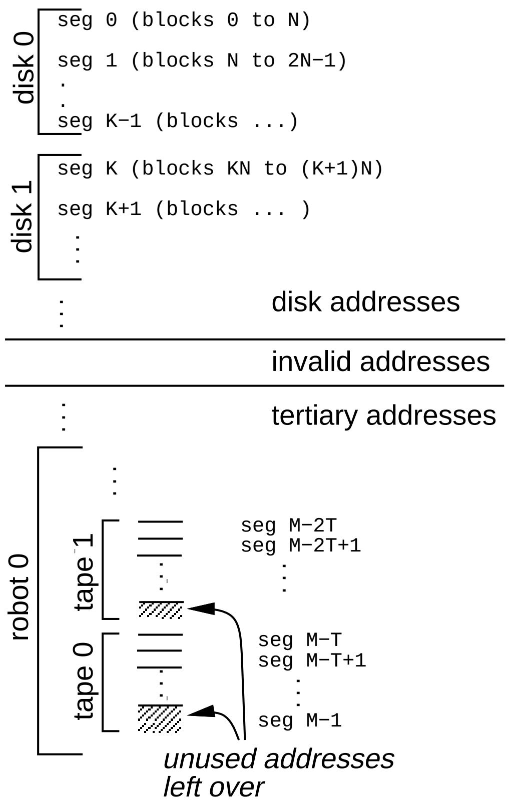

**Figure 4**: Allocation of block addresses to devices in HighLight.

number, the end of the second volume is just below the beginning of the first volume, etc.

The boundary between tertiary and secondary block addresses may be set at any segment multiple. There will likely be a “dead zone” between valid disk and tertiary addresses; attempts to access these blocks results in an error. In principle, the addition of tertiary or secondary storage is just a matter of claiming part of the dead zone by adjusting the boundaries and expanding the file system’s summary tables. However, there currently is no tool to make such adjustments after a file system has been created.

HighLight uses a single block address space for ease of implementation. By using the same format block numbers as the original LFS, HighLight can use much of its code as is. However, with 32-bit block numbers and 4-kilobyte blocks, it is restricted to less than 16 terabytes of total storage. (This limitation is not problematical at the moment, but may prove limiting for future expansion of storage facilities.) One segment's worth of address space is unusable for two reasons: (a) the need for at least one out-of-band block number ("$-1$") to indicate an unassigned block, and (b) the LFS allocates space for boot blocks at the head of the disk, shifting all the segment base addresses up and rendering the last addressable segment too short.

4.4BSD LFS uses a 512-byte block for the partial segment summary, since its block pointers address 512-byte blocks. HighLight block addresses are for 4-kilobyte blocks, so it must use a 4-kilobyte partial segment summary block. Since the summaries include records describing the partial segment, the larger summary blocks could either reduce or increase overall overhead, depending on whether the summaries are completely filled or not. If the summaries in both the original and new versions are completely full, overhead is reduced with the larger summary blocks. In practice, however, HighLight's larger summary blocks are almost always left partially empty. Consuming all the space in the summary block would require that the partial segment contain one block from each of many files (recall from Table 1 that each distinct file with blocks in the partial segment consumes an extra file block description field). This is possible but not likely given the type of files expected in the Sequoia environment.

## Other options considered

The uniform block addressing scheme with 32-bit block addresses split among tertiary and secondary storage arose out of discussions of other potential schemes. The first design considered used a block tertiary address mapping table that would locate file data blocks on tertiary storage by indirecting through the table, indexed by inode-number. Blocks on disk would be fetched in the normal fashion. The size of this table would be prohibitive: assuming a 4-byte entry per tape-resident 4KB block, 1/1024 bytes are needed per block for the table itself; this is far too much overhead for a 9TB tape jukebox.

An alternative design maintained a hash table per tertiary volume to hold a “table of contents,” with entries being tuples of $<$inumber, block range, volume offset$>$. The table would be indexed by hashing the inumber and a portion of the file system block number. The top 8 bits of a 32-bit block number would indicate whether a block was on disk or tertiary storage; one distinguished value for those 8 bits would indicate that the remaining 24 bits should be sliced into a tertiary volume identifier and offset to select the right hash table to locate the desired logical block. On-disk cached blocks would be found by a separate hash table indexed by inumber and logical block number.

Another possibility used a larger block address and segmented it into components directly identifying the device, volume, and offset, and using the device field to dispatch to the appropriate device driver. However, the device/volume identity can just as well be extracted implicitly from the block number by an intelligent device driver that is integrated with the cache. The larger block addresses would also have necessitated many more changes to the base LFS. If the block address were to include both a secondary and tertiary address, the difficulty of keeping disk addresses current when blocks are cached (and updating those disk addresses where they appear in other file system metadata) would seem prohibitive. HighLight instead locates the cached copy of a block by querying a simple hash table indexed by its segment number (which is easily extracted from the block number), and locates the tertiary-resident copy by direct interpretation of the block number.

## 6.4 Secondary Storage Organization

The disks are concatenated by a device driver and used as a single LFS file system. Fresh data are written to the tail of the currently-active log segment. The cleaner reclaims dirty segments by forwarding any live data to the tail of the log. Both the segment selection algorithm, which chooses the next clean segment to be consumed by the log, and the cleaner, which reclaims disk segments, are identical to the 4.4BSD LFS implementations. Unlike the 4.4BSD LFS, though, some of the log segments found on disk are cached segments from tertiary storage.

HighLight's disk space is split into two portions: cache-eligible segments and the remaining segments. A static upper limit (selected when the file system is created) is placed on the number of disk segments that may be in use for caching; after a suitable warm-up time, that many segments would normally be in use for cache space (unless the disk was entirely filled with normal segments). The remainder of the disk is reserved for non-cached segments.

The ifile, containing summaries of segments and inode locations, is a superset of that from the 4.4BSD LFS ifile. It has additional flags available for each segment's summary, such as a flag indicating that the segment is being used to cache a tertiary segment. HighLight also adds an indication of how many bytes of storage are available in the segment (which is useful for bookkeeping for a compressing tape or other container with uncertain capacity) and a cache directory tag.

To record summary information for each tertiary volume, HighLight adds a companion file similar to the ifile. It contains tertiary segment summaries in the same format as the secondary segment summaries found in the ifile. For efficiency of operation, all the special files used by the base LFS and HighLight are known to the migrator and always remain on disk.

As an example of the overhead needed, the ifile requires 1 block for cleaner summary information, 1 block for each 102 on-disk segments, and 1 block for each 341 files. Assuming 10GB of disk space, a 1MB ifile would support over 52,000 files; each additional megabyte would support an additional 87,296 files. The tape summary file requires 1 block for each 102 tertiary volumes, an almost negligible amount for the tertiary devices Sequoia has at hand.

The support necessary for the migration policies may only require user-level support in the migrator, or may involve additional kernel code to record access patterns. Other special state that a migrator might need to implement its policies can be constructed in additional distinguished files. This might include sequentiality extent data (describing which parts of a file are sequentially accessed) or file clustering data (such as a recording of which files are to migrate together).

If a need arises for more disk storage, it is possible to initialize a new disk with empty segments and adjust the file system superblock parameters and ifile to incorporate the added disk capacity. If it is necessary to remove a disk from service, its segments can all be cleaned (so that the data are copied to another disk) and marked as having no storage. Tertiary storage may theoretically be added or removed in a similar way, but there is no current tool to make such changes.

## 6.5 Tertiary Storage Organization

Tertiary storage in HighLight is viewed as an array of devices each holding an array of media volumes, each of which contains an array of segments. Media are currently consumed one at a time by the migration process. The migrator may wish to direct several migration streams to different media, but HighLight does not support that in the current implementation.

The need for tertiary media cleaning should be rare, because the migrator attempts to migrate only stable data, and to have available an appropriate spare capacity in the tertiary storage devices. Indeed, the current implementation does not clean tertiary media. HighLight will eventually have a cleaner for tertiary storage that will clean whole media at a time to minimize the media swap and seek latencies.[3]

Since tertiary storage is often very slow (sometimes with access latencies for loading a volume and seeking to the desired offset running over a minute), the relative penalty of taking a bit more time to access the tertiary storage in return for generality and ease of management of the tertiary storage is an acceptable tradeoff. The tertiary storage is accessed via “Footprint”, a user-level controller process that uses Sequoia’s generic robotic storage interface. It is currently a library linked into the I/O server, but the interface could be implemented by an RPC system to allow the jukebox to be physically located on a machine separate from the file server. This will be important for the future environment due to hardware and device driver constraints. Using Footprint

[3] Minimizing volume insertion and seek passes is also important, as some tapes become increasingly unreliable after too many readings or too many insertions in tape readers.

also simplifies the utilization of multiple types of tertiary devices by providing a uniform interface.

## 6.6 Pseudo Devices

HighLight relies heavily on pseudo device drivers that do not communicate directly with a device but instead provide a device driver *interface* to extended functionality built upon other device drivers and specialized code. For example, a striped disk driver provides a single device interface built on top of several independent disks (by mapping block addresses and calling the drivers for the component disks).

HighLight uses pseudo device drivers for:

- A striping driver to provide a single block address space for all the disks.
- A block cache driver that sends disk requests down to the striping disk pseudo driver and tertiary storage requests to either the cache (which then uses the striping driver) or the tertiary storage pseudo driver.
- A tertiary storage driver to pass requests up to the user-level tertiary storage manager.

Figure 5 shows the organization of the layers. The block map driver, segment cache and tertiary driver are fairly tightly coupled for convenience. The block map pseudo-device handles `ioctl()` calls to manipulate the cache and to service kernel I/O requests, and handles `read()` and `write()` calls to provide the I/O server with access to the disk device to copy segments on or off of the disk.

To handle a demand fetch request, the tertiary driver simply queues it, wakes up a sleeping service process, and then sleeps as usual for any block I/O. The service process directs the I/O process to fetch the data to disk. When the segment has been fetched, the service process completes the block I/O by calling into the kernel with a completion message. The kernel then restarts the original I/O that resulted in the demand fetch; this time it is satisfied by the cache and completes like any normal block I/O. The original process receives its data and continues its execution.

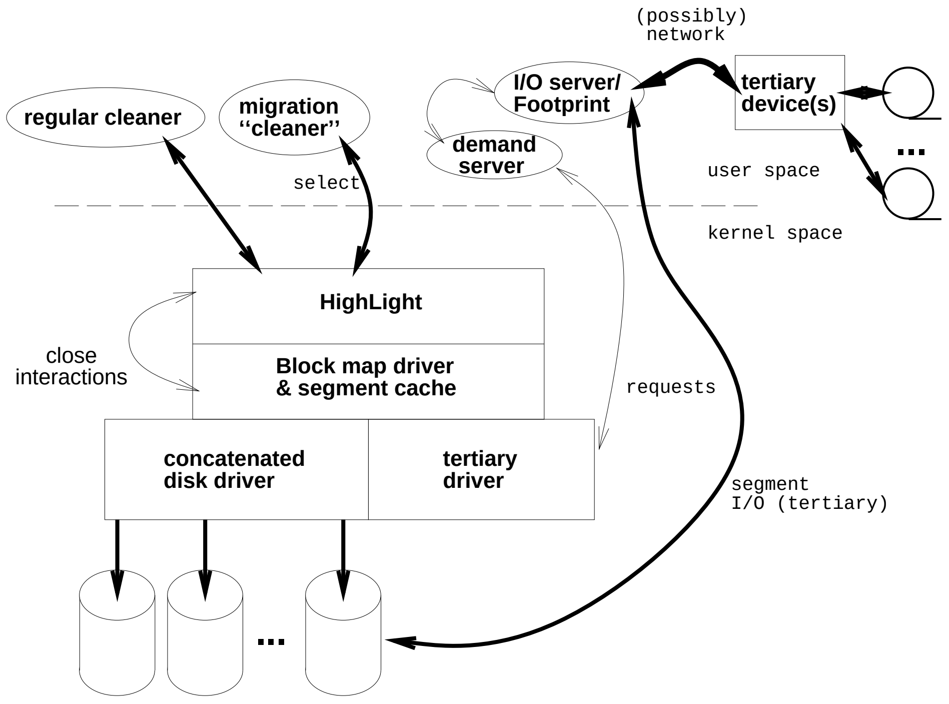

**Figure 5**: The layered architecture of the HighLight implementation. Heavy lines indicate data or data/control paths; thin lines are control paths only.

## 6.7 User level processes

There are three user-level processes used in HighLight that are not present in the regular 4.4BSD LFS: the kernel request service process, the I/O process, and the migrator. The service process waits for requests from either the kernel or from the I/O process: The I/O process may send a status message, while the kernel may request the fetch of a non-resident tertiary segment, the ejection of some cached line (in order to reclaim its space), or a write to tertiary storage of a freshly-assembled tertiary segment.

If the kernel requests a write to tertiary storage or a demand fetch, the service process records the request and forwards it to the I/O server. For a demand fetch of a non-resident segment, the service process selects an on-disk segment to act as the cache line. If there are no clean segments available for that use, the service process selects a resident cache line to be ejected and replaced. When the I/O server replies that a fetch is complete, the service process calls the kernel to complete the servicing of the request. The service process interacts with the kernel via `ioctl()` and `select()` calls on a character special device representing the unified block address space.

The I/O server is spawned as a child of the service process. It waits for requests from the service process, executing each request and replying with a status message. It accesses the tertiary storage device(s) through the Footprint interface, and the on-disk cache directly via a character (raw) pseudo-device. Direct access avoids memory-memory copies (for outgoing segments) and pollution of the block buffer cache with blocks ejected to tertiary storage (of course, after a demand fetch, those needed blocks will eventually end up in the buffer cache). Any necessary raw disk addresses are passed to the I/O server as part of the service process's request.

The I/O server is a separate process primarily to provide for some overlap of I/O with other kernel request servicing. If more overlap is required (e.g. to better utilize multiple tertiary volume reader/writers), the I/O server or service process could be rewritten to farm out the work to several processes or threads to perform overlapping I/O.

The third HighLight-specific process, the migrator, embodies the migration policy of the file system, directing the migration of file blocks to tertiary storage segments. It has direct access to the raw disk device, and may examine disk blocks to inspect inodes, directories, or other structures needed for its policy decisions. It selects file blocks by some criteria, and uses a system call (`lfs_bmapv()`, the same call used by the regular cleaner to determine which blocks in a segment are still valid) to find their current location on disk. If they are indeed on disk, it reads them into memory and directs the kernel (via the `lfs_migratev()` call, a variant of the call the regular cleaner uses to move data out of old segments) to gather and rewrite those blocks into the staging segment on disk. Once the staging segment is filled, the kernel requests the service process to copy the segment to tertiary storage.

To migrate an entire file, the migrator simply uses `lfs_bmapv()` system call to locate all the file's blocks, reads them directly from the disk device into memory, and passes them to the `lfs_migratev()` system call.

### 6.8 Design Summary

HighLight's design incorporates both kernel support and user-level utility support to provide its storage hierarchy management. Applications call the kernel like usual for file service; it gathers assistance from the user-level utilities when needed to service a request or to copy segments to tertiary storage. The user-level utilities provide migration policy (with the kernel mechanism actually preparing tertiary segments) and tertiary storage access (via the Footprint library).

There are some potential problems that this design can generate. Since the file accesses to HighLight seem just like those to other file systems, it is possible that an application may be delayed much longer than expected due to a wait for tertiary storage access. There is no good way for an application to anticipate this, and no way for an application to abort such an access. Also, the performance impact of migration policies (gathering access traces, analysis of the traces or of file status information, bookkeeping, etc.) is not clear, and may be fairly detrimental to performance.

## 7 Performance micro-benchmarks

The benchmarks were designed to answer two basic questions: (a) how do the HighLight modifications to 4.4BSD LFS affect its performance, and (b) does the migration mechanism perform well enough to keep up with the slowest device's transfer speed. The modifications have only a small impact, and in some situations HighLight can get close to the transfer speed of the magneto-optical disk.

The tests ran on an HP 9000/370 CPU with 32 MB of main memory (with 3.2 MB of buffer cache) running 4.4BSD-Alpha. HighLight had a DEC RZ57 SCSI disk drive for the tests, with the on-disk filesystem occupying an 848MB partition. The tertiary storage device was a SCSI-attached HP 6300 magneto-optic (MO) changer with two drives and 32 cartridges. One drive was allocated for the currently-active writing segment, and the other for reading other platters (the writing drive also fulfilled any read requests for its platter). Some of the tests were run for a version of FFS with read- and write-clustering, which coalesces adjacent block I/O operations for better performance. LFS uses the same read-clustering code. To force more frequent volume changes, the tests constrained HighLight's use of each platter to 40MB (since the tests didn't have large amounts of data with which to fill the platters to capacity). The false size of 40MB was used so that the "large objects" (described below) would span more than one volume.

Unfortunately, the autochanger device driver does not disconnect from the SCSI bus, and any media swap transactions "hog" the SCSI bus until the robot has finished moving the cartridges. This is a result of a simple device driver implementation; programmer resources to modify the driver to release the SCSI bus on media change operations were not available. Such media swaps can take many seconds to complete.

## 7.1 Large object performance

The benchmark of Stonebraker and Olson [17] tested HighLight's performance with large "objects", measuring I/O performance on relatively large transfers. To compare HighLight to the basic LFS mechanism, the benchmark ran with three basic configurations:

1. The basic 4.4BSD LFS.

2. The HighLight version of LFS, using files that have not been migrated.

3. The HighLight version of LFS, using migrated files that are all in the on-disk segment cache.

The basic 4.4BSD LFS provides a baseline from which to evaluate HighLight's performance. Using HighLight with non-migrated files lets us examine the extra overhead imposed by its modifications to the basic LFS structures; using it with cached files lets us examine the overhead of the caching mechanism. (There are separate file-access benchmarks for uncached files in the following section.)

The large object benchmark starts with a 51.2MB file, considered a collection of 12,500 frames of 4096 bytes each (these could be database data pages, etc.). The buffer cache is flushed before each operation in the benchmark. The following operations comprise the benchmark:

- Read 2500 frames sequentially (10MB total)
- Replace 2500 frames sequentially (logically overwrite the old ones)
- Read 250 frames randomly (uniformly distributed over the 12500 total frames, selected with the 4.4BSD `random()` function with the sum of the time-of-day and process id as the seed)
- Replace 250 frames randomly
- Read 250 frames with 80/20 locality: 80% of reads are to the sequentially next frame; 20% are to a random next frame.
- Replace 250 frames with 80/20 locality.

Note that for the tests using HighLight with migrated files, any modifications go to local disk rather than to tertiary storage, so that portions of the file live in cached tertiary segments and other portions in regular disk segments. In practice, the migration policies attempt to avoid this situation by migrating only file blocks that are stable.

Table 2 shows measurements for the large object test. This benchmark ran on the plain 4.4BSD-Alpha Fast File System (FFS) as well, using 4096-byte blocks for FFS (the same basic size as used by LFS and HighLight) with the maximum contiguous block count set to 16. With that configuration, FFS tries to allocate file blocks to fill up a contiguous 16-block area on disk, so that it can perform I/O operations with 64-kilobyte transfers.

The base LFS compares unfavorably to the plain FFS for sequential writes; this is most likely due to extra buffer copies performed inside the LFS code (it copies block buffers into a staging

| Phase | FFS | | Base LFS | |
|---|---|---|---|---|
| | time | throughput | time | throughput |
| 10MB sequential read | 10.46 s | 1002KB/s | 12.8 s | 819KB/s |
| 10MB sequential write | 10.0 s | 1024KB/s | 16.4 s | 639KB/s |
| 1MB random read | 6.9 s | 152KB/s | 6.8 s | 154KB/s |
| 1MB random write | 3.3 s | 315KB/s | 1.4 s | 749KB/s |
| 1MB read, 80/20 locality | 6.9 s | 152KB/s | 6.8 s | 154KB/s |
| 1MB write, 80/20 locality | 1.48 s | 710KB/s | 1.2 s | 873KB/s |

| Phase | HighLight (on-disk) | | HighLight (in-cache) | |
|---|---|---|---|---|
| | time | throughput | time | throughput |
| 10MB sequential read | 12.9 s | 813KB/s | 12.9 s | 813KB/s |
| 10MB sequential write | 17.0 s | 617KB/s | 17.6 s | 596KB/s |
| 1MB random read | 6.9 s | 152KB/s | 7.1 s | 148KB/s |
| 1MB random write | 1.4 s | 749KB/s | 1.3 s | 807KB/s |
| 1MB read, 80/20 locality | 6.9 s | 152KB/s | 7.1 s | 148KB/s |
| 1MB write, 80/20 locality | 1.4 s | 749KB/s | 1.4 s | 749KB/s |

**Table 2**: Large Object performance tests. Time values are elapsed times; throughput is calculated from the elapsed time and total data volume. The slight differences between on-disk and in-cache HighLight values may be negligible due to the variation in the measurements between test runs. The FFS measurements are from a version with read and write clustering. For the LFS measurements, the disk had sufficient clean segments so that the cleaner did not run during the tests.

area before writing to disk, so that the disk driver can do a single large transfer). For HighLight, when data have not been migrated to secondary storage, there is a slight performance degradation versus the base LFS (due to the slightly modified system structures). When data have been “migrated” but remain cached on disk, the degradation is small (and may be due to experimental fluctuations between test runs). From these measurements, it appears that HighLight’s modifications do not significantly impact performance if file data are resident on secondary storage, whether they be in a cached tertiary segment or in a regular on-disk segment.

## 7.2 Access Delays

To measure the delays incurred by a process waiting for file data to be fetched into the cache, this test migrated some files, ejected them from the cache, and then read them (so that they were fetched into the cache again). Both the access time for the first byte to arrive in user space and the elapsed time to read the whole files were recorded. The files were read from a newly-mounted filesystem (so that no blocks were cached), using the standard I/O library with an 8KB-buffer. The tertiary volume was in the drive when the tests began, so time-to-first-byte does not include the media swap time. Table 3 shows the first-byte and total elapsed times for disk-resident (both HighLight and FFS) and uncached files. FFS is faster to access the first byte, probably because it fetches fewer metadata blocks (LFS needs to consult the inode map to find the file). The time-to-first-byte is fairly even among file sizes, indicating that HighLight does make file blocks available to user space as soon as they are on disk. The total time for the uncached file read of 10MB is somewhat more than the sum of the measured in-cache access time and the required transfer time (computable from the value in Table 5), indicating some inefficiency in the fetch process. The inefficiency probably stems from the extra copies of demand-fetched segments: they are copied from tertiary storage to memory, thence to raw disk, and are finally re-read through the file system and buffer cache. The implementation of this scheme is simple, but performance suffers. A mechanism to transfer blocks directly from the I/O server memory to the buffer cache might provide substantial improvements, but would involve substantial modifications to the virtual memory subsystems of the operating system. Such modifications were not feasible in the time frame of this project. They would also upset some of the modularity of 4.4BSD, and complicate any porting effort of this implementation.

## 7.3 Migrator throughput

To measure the available bandwidth of the migration path and see if it can keep pace with the transfer rate of the slowest device, the original 51.2MB file from the large object benchmark was

| File size | FFS access times | | HighLight access times | | | |
|---|---|---|---|---|---|---|
| | | | in-cache | | uncached | |
| | First byte | Total | First byte | Total | First byte | Total |
| 10KB | 0.06 s | 0.09 s | 0.11 s | 0.12 s | 3.57 s | 3.59 s |
| 100KB | 0.06 s | 0.27 s | 0.11 s | 0.27 s | 3.59 s | 3.73 s |
| 1MB | 0.06 s | 1.29 s | 0.10 s | 1.55 s | 3.51 s | 8.22 s |
| 10MB | 0.07 s | 11.89 s | 0.09 s | 13.68 s | 3.57 s | 44.23 s |

**Table 3**: Access delays for files, in seconds. The time to first byte includes any delays for fetching metadata (such as an inode) from tertiary storage. The FFS measurements are from a version with read and write clustering.

| Phase | Percentage of time consumed |
|---|---|
| Footprint write | 62% |
| I/O server read | 37% |
| Migrator queuing | 1% |

**Table 4**: A breakdown of the components of the I/O server/migrator elapsed run times while transferring data from magnetic to magneto-optical (MO) disk.

migrated entirely to tertiary storage, while the components of the migration mechanism were timed. This involved the migrator process, which collected the file data blocks and directed the kernel file system to write them to fresh cache segments, the server process, which dispatched kernel requests to copy out dirty cache segments, and the I/O process, which performed the copies.

The migration path measurements are divided into time spent in the Footprint library routines (which includes any media change or seek as well as transfer to the tertiary storage), time spent in the I/O server main code (copying from the cache disk to memory), and queuing delays. Table 4 shows the percentage of real time spent in each phase; the MO disk transfer rate is the main factor in the performance, resulting in the Footprint library consuming the bulk of the running time.

To get a baseline for comparison with HighLight, the raw device bandwidth available was measured by reading and writing with the same I/O sizes as HighLight uses (whole segments).

| I/O type | Performance |
|---|---|
| Raw MO read | 451KB/s |
| Raw MO write | 204KB/s |
| Raw RZ57 read | 1417KB/s |
| Raw RZ57 write | 993KB/s |
| Raw RZ58 read | 1491KB/s |
| Raw RZ58 write | 1261KB/s |
| Volume change | 13.5s |

**Table 5**: Raw device measurements. Raw throughput was measured with a set of sequential 1-MB transfers. Media change measures time from an eject command to a completed read of one sector on the MO platter. The RZ58 read performance may reflect a limitation in the SCSI-I bus rather than the disk drive's maximum transfer rate.

Read tests used `dd`; writes were measured with a simple program to write from memory to disk. A final measurement determined the average time from the start of a volume swap to the replacement volume being ready for reading. Table 5 shows the raw device measurements.

Table 6 shows measurements of two distinct phases of migrator throughput when writing segments to MO disk. The two phases arose in the migration test: the initial phase when the migrator was copying file data to new cache segments and the I/O server was copying dirty segments to tertiary storage; and the second phase when the migrator completed and only the I/O server accessed the cache segments.

The total throughput provided when the magnetic disk is in use simultaneously by the migrator (reading blocks and creating new cached segments) and by the I/O server (copying segments out to tape) is significantly less than the total throughput provided when the only access to the magnetic disk is from the I/O server. When there is no disk arm contention, the I/O server can write at nearly the full bandwidth of the tertiary volume. The magnetic disk and the optical disk shared the same SCSI bus; both were in use simultaneously for the entire migration process. Since both disks were in use for both the disk arm contention and non-contention phases, this suggests that SCSI bandwidth was not the limiting factor and that performance might improve by using a separate disk spindle for the staging cache segments. An additional test using a slower HPIB-connected disk (an

| Phase | Throughput (RZ57) | Throughput (RZ57+RZ58) | Throughput (RZ57+HP7958A) |
|---|---|---|---|
| Magnetic disk arm contention | 111KB/s | 127KB/s | 46.8KB/s |
| No arm contention | 192KB/s | 202KB/s | 145KB/s |
| Overall | 135KB/s | 149KB/s | 99KB/s |

**Table 6**: Migrator throughput measurements for the migration throughput benchmark, showing the phases with and without disk arm contention.

HP7958A) as the staging area showed significant degradation; using a faster SCSI disk (an RZ58) for the staging area showed a modest (almost 15%) improvement in the contention phase. There is still some disk arm contention between the migrator writing out staging segments and the archiver reading those segments, but they do not compete with the disk arm used by the migrator gathering blocks for migration from the regular segments.

### 7.4 Measurement summary

The benchmarks showed that HighLight's modifications to 4.4BSD LFS only slightly reduce its performance (when file data are disk-resident), and that in some cases it can utilize nearly the full transfer speed of the magneto-optical disk.

## 8 Related Work

There are several systems either available or under development that might also provide storage management to meet Sequoia's needs; there are also some studies on migration that may provide insight applicable to HighLight's migration policies. The following subsections survey these works.

## 8.1 Storage systems

Inversion [7] provides file service as an integrated facility of the POSTGRES object-oriented relational database system. Files may be stored and manipulated using the database interface; they may be accessed by using library routines similar to the standard POSIX file-related system calls (`read()`, `write()`, etc.). The POSTGRES system includes a storage manager for the Sony WORM jukebox, and will soon support the Metrum tape jukebox. The no-overwrite storage manager provides a magnetic disk cache to speed access to WORM-resident data.

Unlike HighLight, Inversion currently does not support access to files through the regular operating system interface. Its implementors plan to provide NFS [10] access to its files at some future date. However, Inversion does provide access within the framework of POSTGRES, which is used for storing much of the current Sequoia global-change data. One possibility for future work is to build Inversion on top of HighLight rather than directly on the tertiary devices.

The Jaquith system provides user tools to archive or fetch data from an Exabyte EXB-120 Cartridge Handling Subsystem (a robotic tape changer); it is intended as a supplement to magnetic-disk file storage, and not as a replacement. Users explicitly decide which files to store or fetch; Jaquith handles I/O to and from and cataloging of the tapes and provides a magnetic disk cache for the tape metadata. There are no plans to extend its interface to provide operating system-like file service.

Jaquith serves a different user model than HighLight. It expects users to explicitly identify files to be sent to tertiary storage, with the implicit assumption that users will expect delays in fetching/storing files through Jaquith. HighLight may incur those same delays, but has no way for an the application to predict a delay or determine if a delay is just due to slow access or due to a hardware or software fault.

The StorageServer 100 product from Digital Equipment Corporation provides a magnetic disk front-end for an optical jukebox; files may live on magnetic disk, optical disk, or both. Fixed-size pieces of files (64 kilobyte chunks) may be migrated individually. The system administrator may set parameters for the automatic file migration policies. It appears fairly flexible, but will not completely fulfill Sequoia's capacity needs since it does not support robotic tape jukeboxes.

UniTree [2] provides network access (either NFS or FTP) to its storage arena. It migrates files between magnetic disk, optical disk, robot-loaded tapes, and manual-mount tapes, using a space-time product metric for its migration decisions, coupled with a high-water mark/low-water mark scheme to start and stop the purging process. It copies files from magnetic disk (their first home) to other devices soon after creation, so that disk space may be quickly reclaimed if needed by just deleting enough files from disk. HighLight is similar in those two respects, in that it initially puts files on disk and it copies segments out to tape when they are ready rather than when their disk space is needed.

UniTree supports only whole-file migration between storage devices, while HighLight can support finer-grained migration. It is important to allow such fine-grained migration since POSTGRES will be used for some Sequoia data, and its accesses within a relation (stored in a file) are page-oriented and data-driven. Dormant tuples in a relation should be eligible for migration to tertiary storage; this requires a migration unit finer than the file.

## 8.2 Migration

Some previous studies have considered automatic migration mechanisms and policies for tertiary storage management. Strange [18] develops a migration model based on daily "clean up" computation that migrates candidate files to tertiary storage once a day, based on the next day's projected need for consumable secondary storage space. While Strange provides some insight on possible policies, HighLight should not require a large periodic computation to rank files for migration; instead it allows a migrator process to run continuously, monitoring storage needs and migrating file data as required.

Unlike whole-file migration schemes such as Strange's or UniTree's, HighLight should allow migration of portions of files rather than whole files. A partial-file migration mechanism can support whole file migration, if desired for a particular policy. HighLight also should allow file system metadata, such as directories, inode blocks or indirect pointer blocks, to migrate to tertiary storage. Because it migrates whole segments as a unit, it uses the same indexing structures for all file data, whether they reside on tertiary or secondary storage. It is somewhat wasteful of storage to continue using the full block addressing scheme when file data are laid out in large contiguous chunks on tape. However, using the same format is extremely convenient, since it can just copy segments *in toto* to and from tertiary storage, without needing any data format conversion during the transfer. The convenience and simplicity of this scheme outweighs the potential space savings of using a more compact encoding.

A back-of-the-envelope calculation suggested by Ethan Miller shows why migration of indirect blocks is useful: Assuming 200MB files and a 4K block size, there is an overhead of about 0.1% (200KB) for indirect pointer blocks using the Fast File System (FFS) indirection scheme. A 10TB storage area then requires 10GB of indirect block storage. It seems better to use this 10GB for a cache area instead of wasting it on indirect blocks of files that lay fallow.

Migrating indirect blocks and other metadata may be a two-edged sword, however: while there may be considerable space savings by migrating such blocks, the potential for disaster is greatly increased if a tertiary volume contains metadata required to access some other tertiary volume. If at all possible, policies should arrange that any migrated metadata are contained on the same volume as the data they describe, so that a media failure does not necessitate a traversal of all tertiary media during recovery. If the metadata are self-contained, then there are no “pointers” leaving the volume which could result in inaccessible file data. The only data rendered inaccessible would be those on the failed volume. If the metadata are not self-contained, then a complete traversal of the file system tree (all media volumes) would be needed to reattach or discard any orphaned file blocks, files, or directories which were attached to the file system tree via pointers on the failed volume. This suggests that all migration policies should make vigorous attempts to keep the metadata on volumes self-contained, and perhaps if space allows should just keep metadata on secondary storage.

A final reason why existing systems may not be applicable to Sequoia's needs lies with the expected access patterns. Smith [13, 14] studied file references based mostly on editing tasks; Strange [18] studied a networked workstation environment used for software development in a university environment. Unfortunately, those results may not be directly applicable for the target environment, since Sequoia's file system references will be generated by database, simulation, image processing, visualization, and other I/O intensive-processes [16]. In particular, the database reference patterns will be query-dependent, and will most likely be random accesses within a file rather than sequential access.

HighLight's migration scheme is most similar to that described by Quinlan [8] for the Plan 9 file system. He provides a disk cache as a front for a WORM device that stores all permanent data. When file data are created, their tertiary addresses are assigned but the data are only written to the cache; a nightly conversion process copies that day's fresh blocks to the WORM device. A byproduct of this operation is the ability to "time travel" to a snapshot of the filesystem at the time of each nightly conversion. Unlike that implementation, however, HighLight is designed not to be tied to a single tertiary device and its characteristics (it may wish to reclaim and reuse tertiary storage, which is not possible when using a write-once medium), nor does HighLight provide time travel. Instead it generalizes the 4.4BSD LFS structure to enable migration to and from any tertiary device with sufficient capacity and features.

## 9 Conclusions

Sequoia 2000 needs support for easy access to large volumes of data that won't economically fit on current disks or file systems. HighLight is constructed as an extended 4.4BSD LFS to manage tertiary storage and integrate it into the filesystem, with a disk cache to speed its operation. The extension of LFS to manage a storage hierarchy is fairly natural when segments are used as the transfer unit and as the storage format on tertiary storage.

The mechanisms provided by HighLight are sufficient to support a variety of migration control policies, and provide a good testbed for evaluating these policies. The Footprint interface to tertiary storage made simple the utilization of a non-tape based tertiary storage device, even though HighLight was designed with tape devices as the target. The performance of HighLight's basic mechanism when all blocks reside on disk is nearly as good as the basic 4.4BSD LFS performance. Transfers to magneto-optical tertiary storage can run at nearly the tertiary device transfer speed.

Future work will evaluate the candidate migration policies to determine which one(s) seem to provide the best performance in the Sequoia environment. However, it seems clear that the file access characteristics of a site will be the prime determinant of a good policy. Sequoia's environment may differ sufficiently from others' environments that direct application of previous results may not be appropriate. The architecture is flexible enough to admit implementation of a good policy for any particular site.

## 10 Future Work

To avoid eventual exhaustion of tertiary storage, HighLight will need a tertiary cleaning mechanism that examines tertiary volumes, a task that would best be done with at least two reader/writer devices to avoid having to swap between the being-cleaned volume and the destination volume.

Some other tertiary storage systems do not cache tertiary resident files on first reference, but bypass the cache and return the file data directly. A second reference soon thereafter results in the file being cached. While this is less feasible to implement directly in a segment-based migration scheme, it is achievable in the current implementation by designating some subset of the on-disk cache lines as "least-worthy" and ejecting them first upon reading a new segment. Upon repeated access the cache line would be marked as part of the regular pool for replacement policy (this is essentially a cross between a nearly-MRU cache replacement policy and whatever other policy is in use).

As mentioned above, the ability to add (and perhaps remove) disks and tertiary media while on-line may be quite useful to allow incremental growth or resource reallocation. Constructing such a facility should be fairly straightforward.

There are a couple of reliability issues worthy of further investigation: backup and media failure robustness. Backing up a large storage system such as HighLight would be a daunting effort. Some variety of replication would likely be easier (perhaps having the Footprint server keep two copies of everything written to it). For reliability purposes in the face of media failure, it may be wise to keep critical metadata on disk and back them up regularly, rather than migrate them to a potentially faulty tertiary medium.

It would be nice if the user could be notified about a file access which is delayed waiting for a tertiary storage access. Perhaps the kernel could keep track of a user notification agent per process, and send a “hold on” message to the user.

The cache size is currently fixed statically at file system creation time. A worthwhile investigation would study different dynamic policies for allocating disk space between on-disk and cached segments.

## Acknowledgments

I am quite grateful to the students and faculty in the CS division at Berkeley for their comments on early drafts of this work. An informal study group on Sequoia data storage needs provided many keen insights incorporated in this work. Ann Drapeau helped me understand some reliability and performance characteristics of tape devices. Randy Wang implemented the basic Footprint interface.

I am especially grateful to Carl Staelin, who worked closely with me on this project, to my research advisor Michael Stonebraker for his guidance, to Mendel Rosenblum and John Ousterhout whose LFS work underlies this project, and to Margo Seltzer and the Computer Systems Research Group at Berkeley for implementing 4.4BSD LFS.

# Bibliography

[1] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a Distributed File System. *Operating Systems Review*, 25(5):198–212, October 1991.

[2] General Atomics/DISCOS Division. The UniTree Virtual Disk System: An Overview. Technical report available from DISCOS, P.O. Box 85608, San Diego, CA 92186, 1991.

[3] D. H. Lawrie, J. M. Randal, and R. R. Barton. Experiments with Automatic File Migration. *IEEE Computer*, 15(7):45–55, July 1982.

[4] Marshall Kirk McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[5] Ethan L. Miller, Randy H. Katz, and Stephen Strange. An Analysis of File Migration in a UNIX Supercomputing Environment. In *USENIX Association Winter 1993 Conference Proceedings*, San Diego, CA, January 1993. The USENIX Association. To appear.

[6] James W. Mott-Smith. The Jaquith Archive Server. UCB/CSD Report 92-701, University of California, Berkeley, Berkeley, CA, September 1992.

[7] Michael Olson. The Design and Implementation of the Inversion File System. In *USENIX Association Winter 1993 Conference Proceedings*, San Diego, CA, January 1993. The USENIX Association.

[8] Sean Quinlan. A Cached WORM File System. *Software—Practice and Experience*, 21(12):1289–1299, December 1991.

[9] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. *Operating Systems Review*, 25(5):1–15, October 1991.

[10] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network File System. In *USENIX Association Summer 1985 Conference Proceedings*, Portland, OR, 1985. The USENIX Association.

[11] Margo Seltzer, Keith Bostic, Marshall Kirk McKusick, and Carl Staelin. An Implementation of a Log-structured File System for UNIX. In *USENIX Association Winter 1993 Conference Proceedings*, San Diego, CA, January 1993. The USENIX Association. To appear.

[12] Margo Seltzer and Ozan Yigit. A New Hashing Package for UNIX. In *USENIX Association Winter 1991 Conference Proceedings*, pages 173–184, Dallas, TX, January 1991. The USENIX Association.

[13] Alan Jay Smith. Analysis of Long-Term File Reference Patterns for Application to File Migration Algorithms. *IEEE Transactions on Software Engineering*, SE-7(4):403–417, 1981.

[14] Alan Jay Smith. Long-Term File Migration: Development and Evaluation of Algorithms. *Communications of the ACM*, 24(8):521–532, August 1981.

[15] Michael Stonebraker. The POSTGRES Storage System. In *Proceedings of the 1987 VLDB Conference*, Brighton, England, September 1987.

[16] Michael Stonebraker. An Overview of the Sequoia 2000 Project. Technical Report 91/5, University of California, Project Sequoia, December 1991.

[17] Michael Stonebraker and Michael Olson. Large Object Support in POSTGRES. In *Proc. 9th Int'l Conf. on Data Engineering*, Vienna, Austria, April 1993. To appear.

[18] Stephen Strange. Analysis of Long-Term UNIX File Access Patterns for Application to Automatic File Migration Strategies. UCB/CSD Report 92-700, University of California, Berkeley, Berkeley, CA, August 1992.