HighLight: Using a Log-structured File System for Tertiary Storage Management[*] [†]

John T. Kohl
University of California, Berkeley and
Digital Equipment Corporation

Carl Staelin
Hewlett-Packard Laboratories

Michael Stonebraker
University of California, Berkeley

November 20, 1992

Abstract

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. Log-structured file systems (LFSs) were developed to reduce latencies involved in accessing disk devices, but their sequential write patterns match well with tertiary storage characteristics. Unfortunately, existing versions only manage memory caches and disks, and do not support a broader storage hierarchy.

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 paper 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.

[*] 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 LFS [10], which derives directly from the Sprite Log-structured File System (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. LFS divides the disk into 512KB or 1MB segments, and writes data sequentially within each segment. The segments are threaded together to form a log, so recovery is quick; it entails a roll-forward of the log from the last checkpoint. Disk space is reclaimed by copying valid data from dirty segments to the tail of the log and marking the emptied segments as clean.

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.

HighLight was developed to provide a data storage file system for use by Sequoia researchers. Project Sequoia 2000 [14] 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. HighLight is one of several file management avenues under

exploration as a supporting technology for this research. Other storage management efforts include the Inversion 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 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. We also expect to have a collection of smaller robotic tertiary devices (such as the Hewlett-Packard 6300 magneto-optic changer). HighLight will have exclusive rights to some portion of the tertiary storage space.

HighLight is currently running in our laboratory, with a simple automated file-oriented migration policy as well as a manual migration tool. HighLight can migrate files to tertiary storage and automatically fetch them again from tertiary storage into the cache to enable application access.

The remainder of this paper presents HighLight's mechanisms and some preliminary performance measurements, and speculates on some useful migration policies. We begin with a thumb-nail sketch of the basic Log-structured file system, followed by a discussion of our basic storage and migration model and a comparison with existing related work in policy and mechanism design. We continue with a brief discussion of potential migration policies and a description of HighLight's architecture. We present some preliminary measurements of our system performance, and conclude with a summary and directions for future work.

2. 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* which may be threaded together to form a linear log. New data are appended to the log, and periodically the system checkpoints the state of the system. 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.

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 is necessary for correct operation of the file system, such as whether the segment is clean or dirty, while other information is used to improve the performance of the cleaner, such as the number of live data bytes 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[1].

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, so these index structures are also appended to the log.

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. Making the cleaner 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. 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. The summary also includes a checksum to verify that the entire partial segment is intact on disk and provide an assurance of atomicity. During recovery, the system scans the log, examining each partial segment in sequence.

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

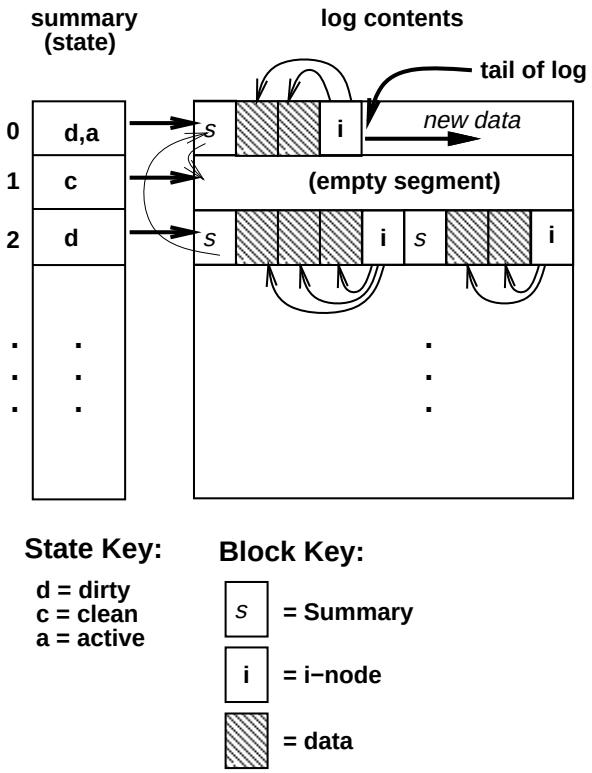

Figure 1: LFS data layout.

Figure 1 shows 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 (data which are still accessible to a user, i.e. not yet deleted or replaced). 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 we have shown 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 both active and dirty. Once Segment 0 fills up the system will begin writing to Segment 1, which is currently clean and empty. Segment 2 was written just before Segment 0; it is dirty and contains live data.

3. Storage and Migration model

HighLight has a "disk farm" to provide rapid access to file data, and one or more tertiary storage devices to provide

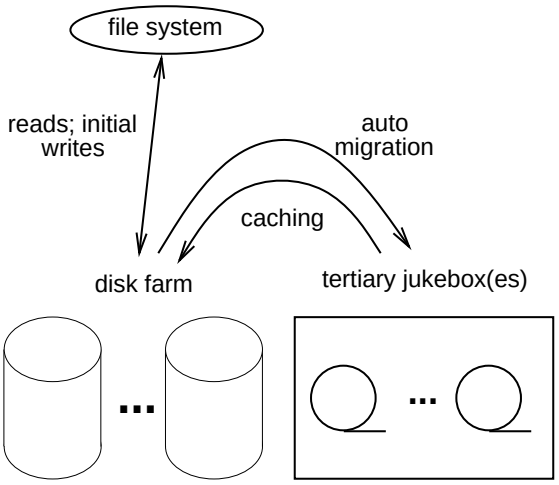

Figure 2: The storage hierarchy.

vast storage. It manages the storage and the migration between the two levels. The basic storage and migration model is illustrated in Figure 2.

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 applying a variant on the cleaning mechanism to provide the migration mechanism. A natural consequence of this layout is the use of LFS segments for the tertiary-resident data representation. 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 a staging segment is filled, it is written to tertiary storage as a unit.

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 non-tertiary 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 cached segments never contain

the sole copy of a block, they may be flushed from the cache at any time if the space is needed for other cache segments or for new data.

3.1. Related work

Some previous studies have considered automatic migration mechanisms and policies for tertiary storage management. Strange [16] develops a migration model based on daily "clean up" computation which 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, we prefer not to require a large periodic migration run (our eventual user base will likely span many time zones, so there may not be any good "dead time" during which to process migration needs); instead we require the ability to run in continuous operation.

Unlike whole-file migration schemes such as Strange's or UniTree's [2], we want to allow migration of portions of files rather than whole files. Our partial-file migration mechanism can support whole file migration, if desired for a particular policy. We also desire to allow file system metadata, such as directories, inode blocks or indirect pointer blocks, to migrate to tertiary storage.[2]

A final reason why existing systems may not be applicable to Sequoia's needs lies with the expected access patterns. Smith [11, 12] studied file references based mostly on editing tasks; Strange [16] studied a networked workstation environment used for software development in a university environment. Unfortunately, those results may not be directly applicable for our environment, since we expect Sequoia's file system references to be generated by database, simulation, image processing, visualization, and other I/O intensive-processes [14]. In particular, the database reference patterns will be query-dependent, and will most likely be random accesses within a file rather than sequential access.

Our 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 which 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

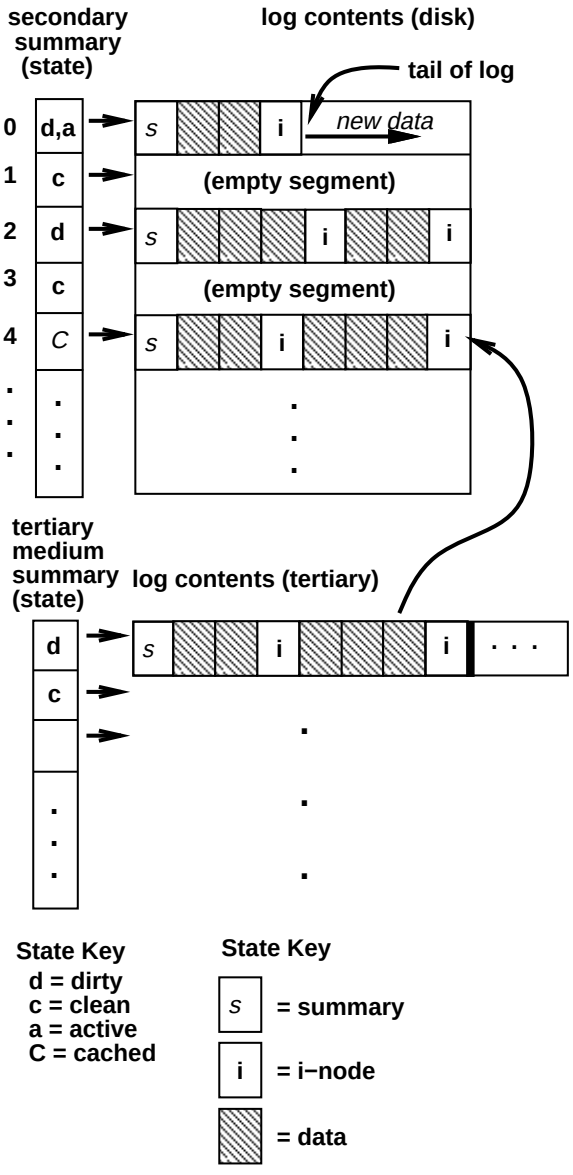

Figure 3: HighLight data layout.

[2] A back-of-the-envelope calculation suggested by Ethan Miller shows why: Assuming 200MB files and a 4K block size, we have an overhead of about 0.1% (200K) for indirect pointer blocks using the FFS indirection scheme. A 10TB storage area then requires 10GB of indirect block storage. Why not use this 10GB for cache area instead of wasting it on indirect blocks of files that lay fallow?

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, we do not wish to be tied to a single tertiary device and its characteristics (we may wish to reclaim tertiary storage), nor do we provide time travel. Instead we generalize the 4.4BSD LFS structure to enable migration to and from any tertiary device with sufficient capacity and features.

The key combination of features which we provide are: the ability to migrate all file system data (not just file contents); tertiary placement decisions made at migration time, not file creation time; data migration in units of LFS segments; migration performed by user-level processes; and migration policy implemented by a user-level process.

4. Migration Policies

Because HighLight includes a storage hierarchy, it must move data up and down the hierarchy. Migration policies may be considered in two parts, writing to tertiary storage, and caching from tertiary storage.

Before describing our migration policies, we must first state our initial assumptions regarding file access patterns, which are based on previous analyses of systems [5, 16, 11]. Our basic assumptions are that file access patterns are skewed, and 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.

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: segment reads. 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 further latencies.

Given perfect predictions, policies should migrate data which provides the best benefit to performance (which could mean something like migrating files which 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. We speculate below on some policies that we will evaluate with HighLight.

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 next section.

4.1. File Size-based rankings

Earlier studies [3, 12] conclude that file size alone does not work well for selecting files as migration candidates; they recommend using a space-time product (STP) ranking metric (time since last access, raised to a small power, times file size). Strange [16] evaluated different variations on the STP scheme for a typical networked workstation configuration. Those three 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 we will evaluate with HighLight.

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.

4.2. 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]. We expect scientific application checkpoints to be read completely and sequentially. 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 are never overwritten [13]. Consequently, block-based information is useful, since old, unreferenced data may migrate to tertiary storage while active data 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. It seems likely that tracking access at a finer grain than whole files can yield a benefit in terms of working set size. 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. We do not yet have a clear implementation strategy for this policy.

4.3. Namespace Locality

When dealing with a collection of small files, it will be more efficient to migrate several related files at once. We can use a file namespace to identify these collections of "related" files, and migrate directory trees or sub-trees 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. Such a tree could be considered in the aggregate as a file for purposes of applying a migration metric (such as STP).

Assuming such a tree 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 tree cluster.

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

4.4. Rewriting Cached Segments

It may be the case that data access patterns to tertiary-backed storage will change over time (for example, if several satellite-collected 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) (perhaps to move segments to different tertiary media with access characteristics more suited to those segments). This reorganization can be accomplished by writing cached segments to a new storage location on the tertiary device while the segment is in the cache.

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 need to be fast reclaims).

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. Such information could be maintained by the process servicing demand fetch requests and shared with the migrator.

4.5. Supporting migration policies

To summarize, we can envision uses for (at least) the following mechanism features in an implementation:

- Basic migration bookkeeping (cache lookup control, data movement, etc.)
- Whole-file migration
- Directory and 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 design and implementation of HighLight, which covers many (but not all) of these desired features.

5. 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. Our discussion here covers HighLight's basic components, block addressing scheme, secondary

and tertiary storage organizations, migration mechanism, and implementation details.

5.1. Components

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

- A second cleaner, called the *migrator*, which collects data for migration from secondary to tertiary storage
- A disk-resident segment cache to hold read-only copies of tertiary-resident segments and request I/O from the user-level processes, implemented as a pseudo-disk driver
- A pseudo-disk driver which 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.).

5.2. 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 32-bit block pointers. 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 medium).

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 medium. Those blocks are then assembled in a "staging segment" addressed by new block numbers assigned to a tertiary medium. 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 served asynchronously, so that the migration control policies may choose to move multiple segments in a single logical operation for transfer efficiency.

Disk segments can be used to cache tertiary segments. Since the cached segments are read-only copies of the tertiary-resident version, cache management is relatively simple (involving no write-back issues). As in the normal LFS, when file data are updated, a new copy of the changed data are appended to the current on-disk log segment; the old copy remains undisturbed until its segment is cleaned or ejected from the cache. We don't clean cached segments on disk; any cleaning of tertiary-resident segments would be done directly with the tertiary-resident copy.

If a process requests I/O on a file for which some necessary metadata or file data are not on secondary storage, the cache 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 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.

5.3. Block addresses

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 medium (disk device, tape cartridge, or jukebox platter) and the segment's location within the medium. The offset identifies a particular block in that segment.

HighLight allocates a fixed number of segments to each tertiary medium. 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 medium is expected to hold. HighLight can tolerate

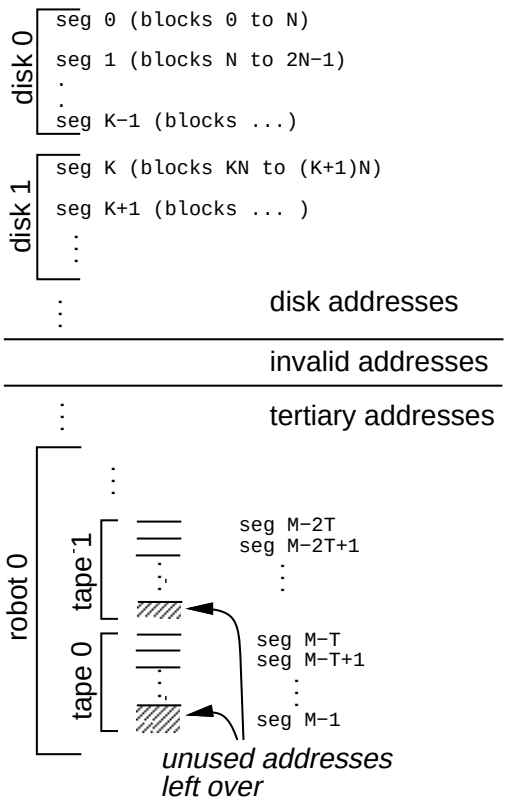

Figure 4: Allocation of block addresses to devices in HighLight.

device-based compression on tertiary storage since it can keep writing segments to a medium until the drive returns an "end-of-tape" message, at which point the medium is marked full and the last (partially written) segment is re-written onto the next tape. If the compression factor exceeds the expectations, however, all the segments will fit on the tape and some storage at its end may be wasted.

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 medium is at the largest block number, the end of the second medium is just below the beginning of the first medium, 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, we do not currently have a tool to make such adjustments after a file system has been created.

We use a single block address space for ease of implementation. By using the same format block numbers as the original LFS, we can use much of its code as is. However, with 32-bit block numbers and 4-kilobyte blocks, we are restricted to less than 16 terabytes of total storage. One segment's worth of address space is unusable for two reasons: (a) we need 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.

We considered using a larger block address and segmenting it into components directly identifying the device, medium, and offset, and using the device field to dispatch to the appropriate device driver. However, the device/medium identity can just as well be extracted implicitly from the block number by an intelligent device driver which is integrated with the cache. The larger block addresses would also have necessitated many more changes to the base LFS, a task which we declined. We considered having the block address include both a secondary and tertiary address, but the difficulty of keeping disk addresses current when blocks are cached (and updating those disk addresses where used) seemed prohibitive. We instead chose to locate the cached copy of a block by querying a simple hash table indexed by segment number.

Using 4-kilobyte blocks necessitates an increased partial segment summary block size (it is only 512 bytes in 4.4BSD LFS). 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. However, the larger summary blocks are almost always too large to be filled in practice, since doing so would require a segment summary to cover an entire segment, and that segment would need to be filled with one block from each of many files. This is possible but not likely given the type of files we expect to find in our environment.

5.4. 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 read-only cached segments from tertiary storage.

The ifile, which contains 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 and should not be cleaned or overwritten. We also add 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).

To record summary information for each tertiary medium, 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.

Other special support which 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). 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.

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.

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.

5.5. 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. We expect that the migrator may wish to direct several migration streams to different media, but do not support that in our current implementation.

We expect the need for tertiary media cleaning to be rare, because we make efforts to migrate only stable data, and to have available an appropriate spare capacity in our tertiary storage devices. Indeed, the current implementation does not clean tertiary media. We will eventually have a cleaner for tertiary storage, which 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 medium and seeking to the desired offset running over a minute), the relative penalty of taking a bit more access time to the tertiary storage in return for generality and ease of management of the tertiary storage access path is an acceptable tradeoff. Our tertiary storage is accessed via "Footprint", a user-level controller process which 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 our environment due to hardware and device driver constraints. Using Footprint also simplifies our utilization of multiple types of tertiary devices, by providing a uniform interface.

5.6. Pseudo Devices

HighLight relies heavily on pseudo device drivers, which 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 respective disks).

HighLight uses pseudo device drivers for:

- A striping driver to provide a single block address space for all the disks.

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

- A block cache driver which sends disk requests down to the striping disk pseudo driver, and which sends 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 enqueues 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 it has been fetched, the service process completes the block I/O by calling into the kernel and restarting the I/O through the cache. It completes like any normal block I/O and wakes up the original process.

5.7. 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 the transfer to tertiary storage of a freshly-written tertiary segment.

If the kernel requests a "push" 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 a request from the service process, executes the request, and replies with a status message. It accesses the tertiary storage device(s) through the Footprint interface, and the on-disk cache directly via the cache raw device. Direct access avoids memory-memory copies 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, 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()`) 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 posts a request of the service process to copy the segment to tertiary storage.

6. Performance micro-benchmarks

To understand and evaluate the performance of HighLight and the impact of our modifications to the basic LFS mechanism, we ran benchmarks with three basic configurations:

1. The basic 4.4BSD LFS.
2. The HighLight version of LFS, using files which have not been migrated.
3. The HighLight version of LFS, using migrated files which are all in the on-disk segment cache.

We ran the tests on an HP 9000/370 CPU with 32 MB of main memory (with 3.2 MB of buffer cache) running 4.4BSD-Alpha. We used a DEC RZ57 SCSI disk drive for our tests, with the on-disk filesystem occupying an 848MB partition. Our tertiary storage device was a SCSI-attached HP 6300 magneto-optic (MO) changer with two

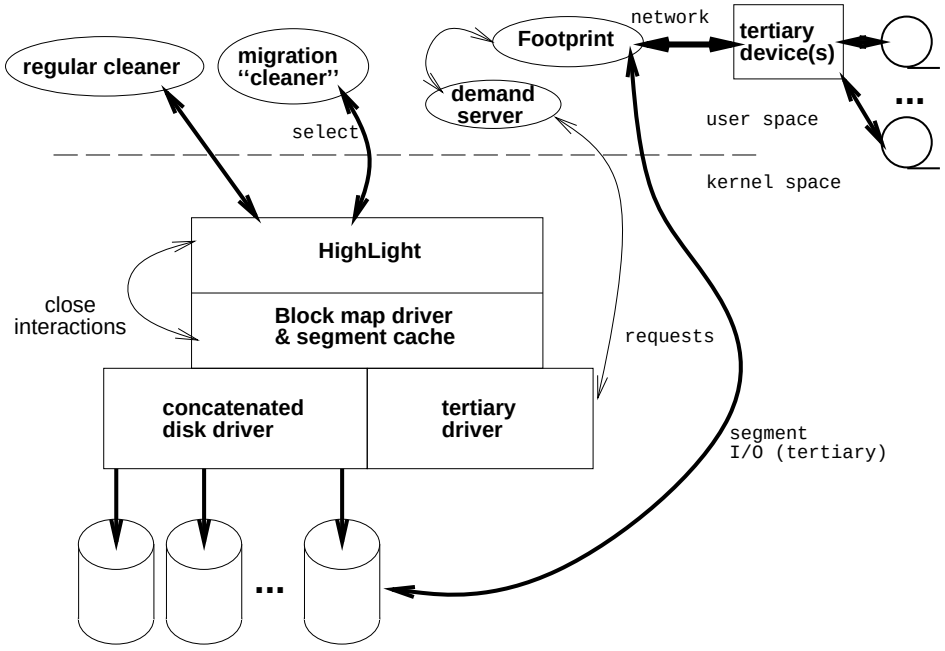

Figure 5: The layered architecture of the HighLight implementation. Heavy lines indicate data or data/control paths; thin lines are control paths only.

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). When running tests with storage To force more frequent medium changes, we constrained HighLight's use of each platter to 40MB (since we didn't have large amounts of data with which to fill the platters to capacity).

Unfortunately, our 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. Such media swaps can take many seconds to complete.

6.1. Large object performance

To test performance with large "objects", we used the benchmark of Stonebraker and Olson [15] to measure I/O performance on relatively large transfers. It starts with a 51.2MB file, considered a collection of 12,500 frames of 4096 bytes each (these could be database data pages, compressed images in an animation, etc). The buffer cache is flushed before each phase of 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 time-of-day 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 HighLight version 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, our migration policies attempt to avoid this situation by migrating only file blocks which are stable.

Phase	FFS		Base LFS		HLFS (on-disk)		HLFS (in-cache)	
	time	throughput	time	throughput	time	throughput	time	throughput
10MB sequential read	10.46 s	1002KB/s	12.8 s	819KB/s	12.9 s	813KB/s	12.9 s	813KB/s
10MB sequential write	10.0 s	1024KB/s	16.4 s	639KB/s	17.0 s	617KB/s	17.6 s	596KB/s
1MB random read	6.9 s	152KB/s	6.8 s	154KB/s	6.9 s	152KB/s	7.1 s	148KB/s
1MB random write	3.3 s	315KB/s	1.4 s	749KB/s	1.4 s	749KB/s	1.3 s	807KB/s
1MB read, 80/20 locality	6.9 s	152KB/s	6.8 s	154KB/s	6.9 s	152KB/s	7.1 s	148KB/s
1MB write, 80/20 locality	1.48 s	710KB/s	1.2 s	873KB/s	1.4 s	749KB/s	1.4 s	749KB/s

Table 1: Large Object performance tests. Time values are elapsed times; throughput is calculated from the elapsed time and total data volume. 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.

File size	FFS access times		HLFS 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 2: 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.

Table 1 shows our measurements for the large object test. We were able to test this benchmark on the plain 4.4BSD-Alpha Fast File System (FFS) as well; we used 4096-byte blocks for FFS (the same basic size as used by LFS and HighLight) with the maximum contiguous block count set to 16 (to result in 64-kilobyte transfers in the best case). The base LFS compares unfavorably to the plain FFS; this is most likely due to extra buffer copies performed inside the LFS code. 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). Even when data have been "migrated" but remain cached on disk, the degradation is small.

6.2. Access Delays

To measure the delays incurred by a process waiting for file data to be fetched into the cache, we migrated some files, ejected them from the cache, and then read them (so that they were fetched into the cache again). We timed both the access time for the first byte to arrive in user space, and the elapsed time. 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 medium was in the drive when the tests began, so time-to-first-byte does not include the media swap time. Table 2 shows the first-byte and total elapsed times for disk-resident (both HLFS 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 in-cache 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.

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

Table 3: A breakdown of the components of the archiver/migrator elapsed run times while transferring data from magnetic to magneto-optical (MO) disk.

I/O type	Performance
Raw MO read	451KB/s
Raw MO write	204KB/s
Raw RZ57 read	1417KB/s
Raw RZ57 write	989KB/s
Media change	13.5s

Table 4: 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.

Phase	Throughput
Magnetic disk arm contention	111KB/s
No arm contention	192KB/s
Overall	135KB/s

Table 5: Migrator throughput measurements for phases with and without disk arm contention.

6.3. Migrator throughput

To measure the available bandwidth of the migration path, we took the original 51.2MB file from the large object benchmark and migrated it entirely to tertiary storage, while timing the components of the migration mechanism. 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 3 shows the measurements; 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, we measured the raw device bandwidth available by using `dd` with the same I/O sizes as HighLight uses (whole segments). We also measured the average time from the start of a medium swap to medium ready for reading. Table 4 shows our raw device measurements.

Table 5 shows our measurements of two distinct phases of migrator throughput when writing segments to MO disk. 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 medium. 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 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.

7. Conclusions

Sequoia 2000 needs support for easy access to large volumes of data which won't economically fit on current disks or file systems. We have constructed HighLight as an extended 4.4BSD LFS. It manages tertiary storage and integrates it into the filesystem, with a disk cache to speed its operation. The mechanisms provided by HighLight are sufficient to support a variety of potential migration control policies, and provide a good testbed for evaluating these policies. 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.

We intend to evaluate our candidate 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. Our architecture is flexible enough to admit implementation of a good policy for any particular site.

8. Future Work

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

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, we could designate some subset of the on-disk cache lines as "least-worthy" and eject 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 study: 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 a medium failure, it may be wise to keep certain metadata on disk and back them up regularly, rather than migrate them to a potentially faulty tertiary medium. Doing so might avoid the need to examine all the tertiary media in order to reconstruct the filesystem after a tertiary medium failure.

Code Availability

Source code for HighLight will be available via anonymous FTP from `postgres.berkeley.edu` when the system is robust enough for external distribution.

Acknowledgments

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

We are especially grateful 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.

References

[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.

[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] 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.

[11] 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.

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

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

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

[15] 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.

[16] 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.

Author Information

John Kohl is a Software Engineer with Digital Equipment Corporation. He holds an MS in Computer Science from the University of California, Berkeley (December 1992) and a BS in Computer Science and Engineering from the Massachusetts Institute of Technology (May 1988). He worked several years at MIT's Project Athena, where he led the Kerberos V5 development effort. He may be reached by e-mail at `jtkohl@cs.berkeley.edu`, or by surface mail at Digital Equipment Corporation, ZKO3-3/U14, 110 Spit Brook Road, Nashua, NH 03062-2698.

Carl Staelin works for Hewlett-Packard Laboratories in the Berkeley Science Center and the Concurrent Systems Project. His research interests include high performance file system design, and tertiary storage file systems. As part of the Science Center he is currently working with Project Sequoia at the University of California at Berkeley. He received his PhD in Computer Science from Princeton University in 1992 in high performance file system design. He may be reached by e-mail at `staelin@hpl.hp.com`, or by telephone at (415) 857-6823, or by surface mail at Hewlett-Packard Laboratories, Mail Stop 1U-13, 1501 Page Mill Road, Palo Alto, CA 94303.

Michael Stonebraker is a professor of Electrical Engineering and Computer Science at the University of California, Berkeley, where he has been employed since 1971. He was one of the principal architects of the INGRES relational data base management system which was developed during the period 1973-77. Subsequently, he constructed Distributed INGRES, one of the first working distributed data base systems. Now (in conjunction with Professor Lawrence A. Rowe) he is developing a next generation DBMS, POSTGRES, that can effectively manage not only data but also objects and rules as well.

He is a founder of INGRES Corp (now the INGRES Products Division of ASK Computer Systems), a past chairman of the ACM Special Interest Group on Management of Data, and the author of many papers on DBMS technology. He lectures widely, has been the keynote speaker at recent IEEE Data Engineering, OOPSLA, and Expert DBMS Conferences, and has been a consultant for several organizations including Citicorp, McDonnell-Douglas, and Pacific Telesis. He may be reached by e-mail at `mike@cs.berkeley.edu` or surface mail at 549 Evans Hall, Berkeley, CA 94720.