# The Episode File System

*Sailesh Chutani*
*Owen T. Anderson*
*Michael L. Kazar*
*Bruce W. Leverett*
*W. Anthony Mason*
*Robert N. Sidebotham*
*Transarc Corporation*

## Abstract

We describe the design of Episode,™ a highly portable POSIX-compliant file system. Episode is designed to utilize the disk bandwidth efficiently, and to scale well with improvements in disk capacity and speed. It utilizes logging of meta-data to obtain good performance, and to restart quickly after a crash.

Episode uses a layered architecture and a generalization of files called *containers* to implement *filesets*. A fileset is a logical file system representing a connected subtree. Filesets are the unit of administration, replication, and backup in Episode.

The system works well, both as a standalone file system and as a distributed file system integrated with the OSF's Distributed Computing Environment (DCE). Episode will be shipped with the DCE as the Local File System component, and is also exportable by NFS. As for performance, Episode meta-data operations are significantly faster than typical UNIX Berkeley Fast File System implementations due to Episode's use of logging, while normal I/O operations run near disk capacity.

## Introduction

This paper describes the Episode™ file system, the local file system for the OSF Distributed Computing Environment (DCE). Episode was intended as a file system for distributed file servers, and is designed to be exported by various network file systems, especially the OSF DCE's Distributed File Service (DFS).

Episode separates the concepts of disk storage and logical file system structure, and provides a number of features not found in most UNIX® file systems, such as those based on the Berkeley Fast File System [MCK 84]. In particular, Episode provides POSIX-style (Draft 11) access control lists, a useful form of replication for slowly changing data, data representations that support storage files of size $2^{32}$ *fragments* (at least $2^{42}$ bytes), and logging techniques that reduce post-crash recovery time and improve the performance of operations that update meta-data. This paper explains the overall architecture of the file system.

## Background

As part of the design process for AFS®4 (which became the Distributed File System component of the DCE), the Episode design team looked at the AFS 3 [SAT 85] file system's file server. Two significant features of AFS 3 were viewed as valuable to preserve for Episode: *access control lists* and AFS 3 *volumes* — which were renamed *filesets*.

Access control lists are valuable in large distributed systems primarily because of the size of the

user community in such systems. In such a large community, users require a flexible mechanism to specify exactly who should be able to access their files. The more traditional UNIX protection mechanism of grouping everyone into one of three categories is often insufficient to express flexible controls on data. While AFS 3 provides ACLs only on directories, Episode provides ACLs on both files and directories, thereby enabling POSIX 1003.6 compliance.

AFS 3 volumes support the separation of disk block storage from the concept of logical file system structure, so that a single pool of disk blocks can provide storage to one, or thousands of separate file system hierarchies [SID 86]. In Episode, each logical file system contains its own *anode* table, which is roughly the equivalent of a Berkeley Fast File System's (BSD) inode table [MCK 84]. Various anodes within a fileset describe its root directory, as well as subsidiary files and directories. Each fileset is independently mountable, and — when a distributed file system is present — independently exportable.[1]

The data representation of filesets facilitates their movement from one partition to another with minimum disruption, even while they are exporting data in a distributed file system. All data within a fileset can be located by simply iterating through the anode table, and processing each file in turn. Furthermore, a file's low-level identifier, which is used by distributed file systems and stored in directories, is represented by its index in the fileset's anode table. This identifier remains constant even after moving a fileset to a different partition or machine.

The general model for resource reallocation in the Episode design is to keep many filesets on a single partition. When a partition begins to fill up, becomes too busy, or develops transient I/O errors, an administrator can move filesets transparently to another partition while allowing continuous access by network and even local clients. Tools are provided to facilitate this move across multiple disks (or multiple servers, using the OSF's DCE). Note that this model of resource reallocation requires the ability to put more than one fileset on a single partition; without this, the only resource reallocation operations available are equivalent to the exchanging of file system contents between partitions, a move of limited utility.

Episode's implementation of fileset moving, as well as other administrative operations, depend upon a mechanism called *fileset cloning*. A fileset clone is a fileset containing a snapshot of a normal fileset, and sharing data with the original fileset using copy-on-write techniques. A cloned fileset is read-only, and is always located on the same partition as the original read-write fileset. Clones can be created very quickly, essentially without blocking access to the data being cloned. This feature is very important to the administrative operations' implementation: the administrative tools use clones instead of the read-write data for as much of their work as possible, greatly reducing the amount of time they require exclusive access to the read-write data.

Episode's underlying disk block storage is provided by *aggregates*. Aggregates [KAZ 90] are simply partitions augmented with certain operations, such as those to create, delete and enumerate filesets.

In a conventional BSD file system, one of the biggest practical constraints on how much disk space a file server can hold is how long the disk check program *fsck* [KOW 78] would run in the event of a crash. Episode uses logging techniques appropriated from the database literature [HAE 83, HAG 87, CHA 88] to guarantee that after a crash, the file system meta-data (directories, allocation bitmaps, indirect blocks and anode tables) are consistent, generally eliminating the need for running "fsck."

This idea is not new. The IBM RS/6000's local file system, JFS [CHA 90], uses a combination of operation logging for the allocation bitmap and new value-only logging for other meta-data. Hagmann followed a similar approach in building a log-based version of the Cedar file system [HAG 87]. On the RS/6000, JFS also uses hardware lock bits in the memory management hardware to determine which records should be locked in memory mapped transactional storage. This technique was earlier supported by the IBM RT/PC's memory mapping unit, although on that system it was not used for a commercially available file system [CHA 88]. Veritas Corporation's VxFS [VER 91] apparently also uses new value-only logging technology. Another system using logging technology is the Sprite LFS [ROS 90], in which all the

[1] In principle at least; at present, the DCE tools only allow the exporting of all of the filesets in a partition.

data is stored in a log. LFS uses operation logging to handle directory updates, and new value-only logging for other operations.

## Data Architecture

The central conceptual object for storing data in Episode is a *container*. A container is an abstraction built on top of the disk blocks available in an aggregate. It is a generalization of a file that provides read, write, create and truncate operations on a sequence of bytes. Containers are described by *anodes*, 252 byte structures analogous to BSD inodes [LEF 89], and are used to store all of the user data and meta-data in the Episode file system.

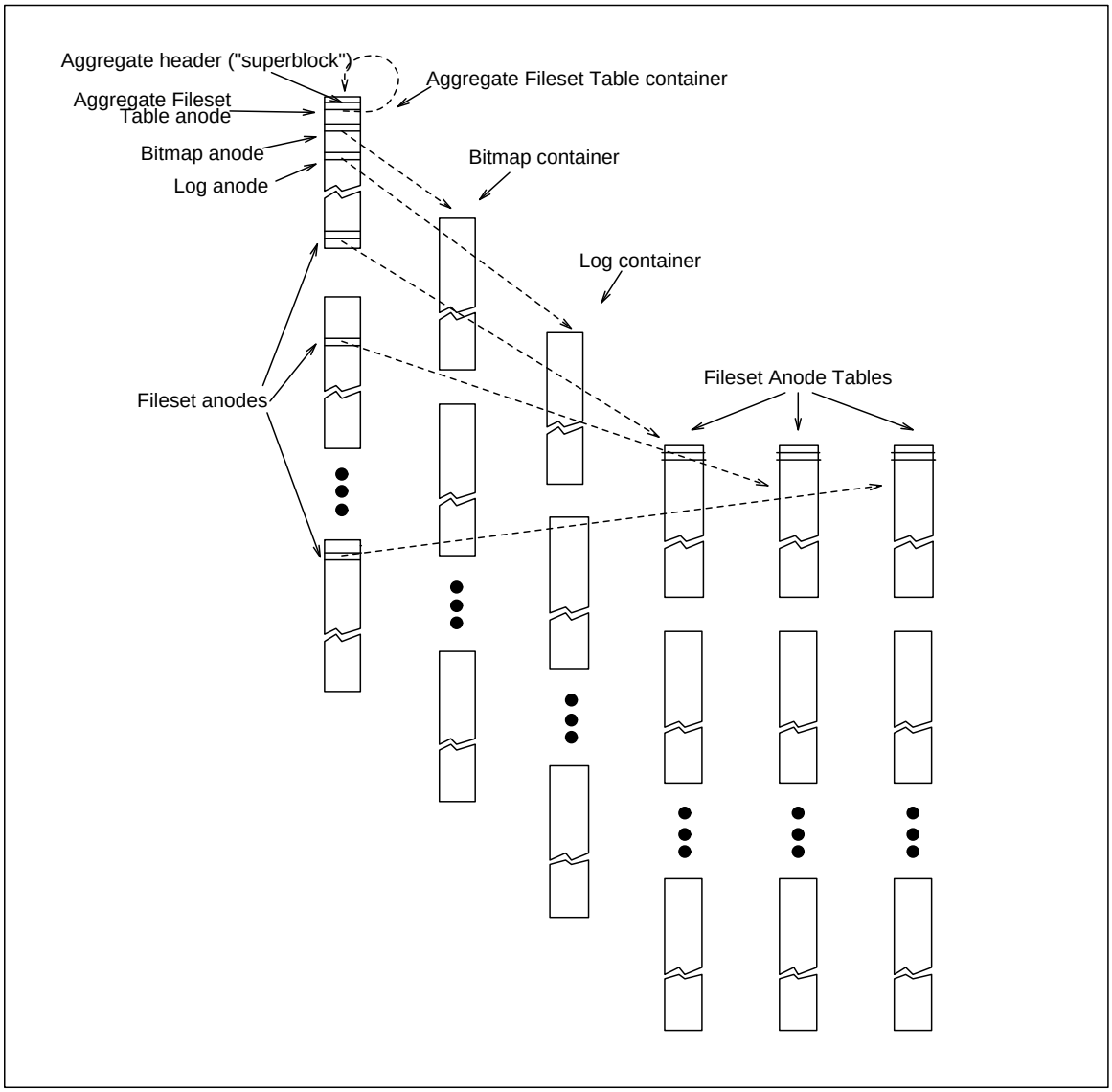


Figure 1: Bird's-eye view of an Episode Aggregate.

A bird's-eye view of an aggregate is provided in Figure 1. Each of the rectangular blocks in the figure represents a file system block, and vertical columns of these blocks represent *containers*. Each Episode aggregate has three specialized containers, the *Bitmap container*, the *Log container*, and the *Aggregate Fileset Table*.

The bitmap container stores two pieces of information about each fragment in the aggregate: whether the fragment is allocated, and whether the fragment represents logged or unlogged data. This last distinction is necessary because certain buffer pool operations have to be performed when reusing a logged block as an unlogged block, and vice versa.

The aggregate fileset table is organized as an array of anodes, one for each fileset in the aggregate. The anode corresponding to a particular fileset describes that fileset's anode table, which is roughly equivalent to a file system's inode table in a BSD file system. An Episode fileset's anode table contains individual anodes describing that fileset's directories, files, symbolic links and access control lists.

References to file system anodes generally come from two sources: names found in directories, and file IDs arriving via network file systems. These references name an anode by its fileset ID and its index within the fileset's anode table. Thus, a reference to a particular anode within a fileset starts by searching the aggregate's fileset table for the desired fileset. Once found, the fileset's anode table container contains an array of anodes, and the specified anode within the fileset is simply selected by its index. In typical operation, all of these steps are significantly sped up by caching.

The log container provides the storage allocated for the aggregate's transaction log. All meta-data updates are recorded in this log. The log is processed as a circular buffer of disk blocks, with the tail of the log stored in memory and forced to disk only when necessary. The log is not actually constrained to be on the same aggregate as the data that it is logging, but this restriction is currently imposed by our initialization utilities.

Containers provide a uniform mechanism for data storage in Episode. All the disk data abstractions in Episode, including the allocation bitmap, the transaction log, the fileset table, all of the individual filesets' anode tables, and all directories and files are stored in containers. Because containers can dynamically grow and shrink, all meta-data allocated to containers can, in principal, be dynamically resized. For example, there is no need for a static allocation of anodes to an individual fileset, since a fileset's anode table container can simply grow if a large number of files are created within that fileset. In addition, since the container abstraction is maintained by one piece of code, the logic for allocating meta-data exists in only one place.

Despite the potential for dynamic resizing all of the meta-data stored in containers, certain containers do not, in the current implementation, change dynamically. The log container does not grow or shrink under normal system operation, since the information that ensures that the log is always consistent would have to be placed in the same log whose size is changing. The partition's allocation bitmap is created by the Episode equivalent of "newfs," but does not change size afterwards. Finally, directories never shrink, except when truncated as part of deletion.

As mentioned above, a fileset clone is a read-only snapshot of a read-write fileset, implemented using copy-on-write techniques, and sharing data with the read-write fileset on a block-by-block basis. Episode implements cloning by cloning each of the individual anodes stored in that fileset. When an anode is initially cloned, both the original writable version of the anode and the cloned anode point to the same data block(s), but the disk addresses in the original anode, both for direct blocks and indirect blocks, are tagged as copy-on-write (COW), so that an update to the writable fileset does not affect the cloned snapshot. When a copy-on-write block is modified, a new block is allocated and updated, and the COW flag in the pointer to this new block is cleared. The formation of clones is illustrated in Figure 2.

## Component Architecture

Episode has the layered architecture illustrated in Figure 3. The operating system independent layer (not shown in the diagram), and the asynchronous I/O (async) layer comprise the portability layers of the system. The operating system independent layer provides system-independent synchronization and timing primitives. The async layer acts as a veneer over the device drivers, hiding small but significant differences in the interfaces between various kernels. It also provides a simple event mechanism, whose

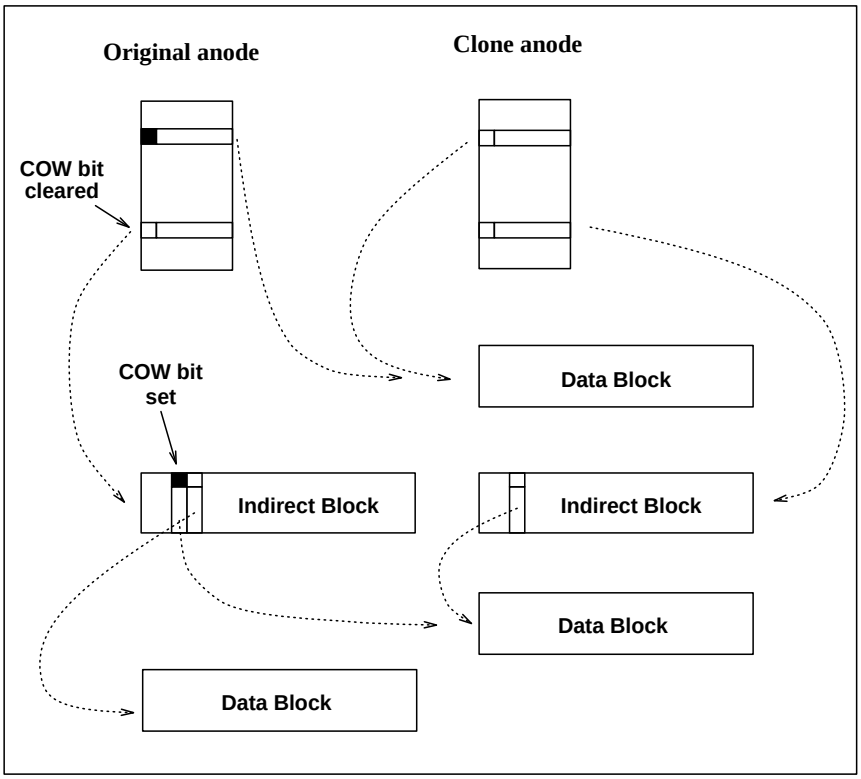


Figure 2: A Container: After Cloning and Extending.

primary purpose is providing operations for awaiting I/O completion events.

Above these base layers is the log/buffer package. This package provides an abstraction very much like the Unix buffer pool, buffering blocks from the disk and writing them as requested. This package also mediates all buffer modifications so that they can be logged as required by the logging strategy employed [HAE 83, MOH 89].

In Episode, all the updates to the meta-data are grouped into *transactions* that are *atomic*, meaning that either all the updates within a transaction (if a transaction *commits*), or none of them (if a transaction *aborts*), will be applied. By making all file system meta-data modifications within atomic transactions, the file system can be restored to a consistent state after a crash.

Episode implements atomic transactions through a a combination of *write-ahead* and *old value/new value* logging techniques [MOH 89]. In a nutshell, this form of logging works by logging, for every update made to any file system meta-data, both the original and new values of the updated data. Furthermore, before the buffer package allows any dirty meta-data buffer to be written back out to the disk, it writes out these log entries to the disk. In the event of a crash, only some of the updates to the file system meta-data may have made it to the disk. If the transaction aborted, then there is enough information in the log to undo all of the updates made to the meta-data, and restore the meta-data to its state before the transaction started. If the transaction committed, there is enough information in the log to redo all of the meta-data updates, even those that hadn't yet made it from the disk buffers to the disk.

The *recovery* procedure runs after a crash, replaying the committed transactions and undoing the uncommitted transactions, and thus restoring the file system to a consistent state. Since the log only contains information describing transactions still in progress, recovery time is proportional to the activity at the time of the crash, not to the size of the disk. The result is that the log-replaying operation runs orders of magnitude faster than the BSD fsck program. There are some cases in which the recovery procedure

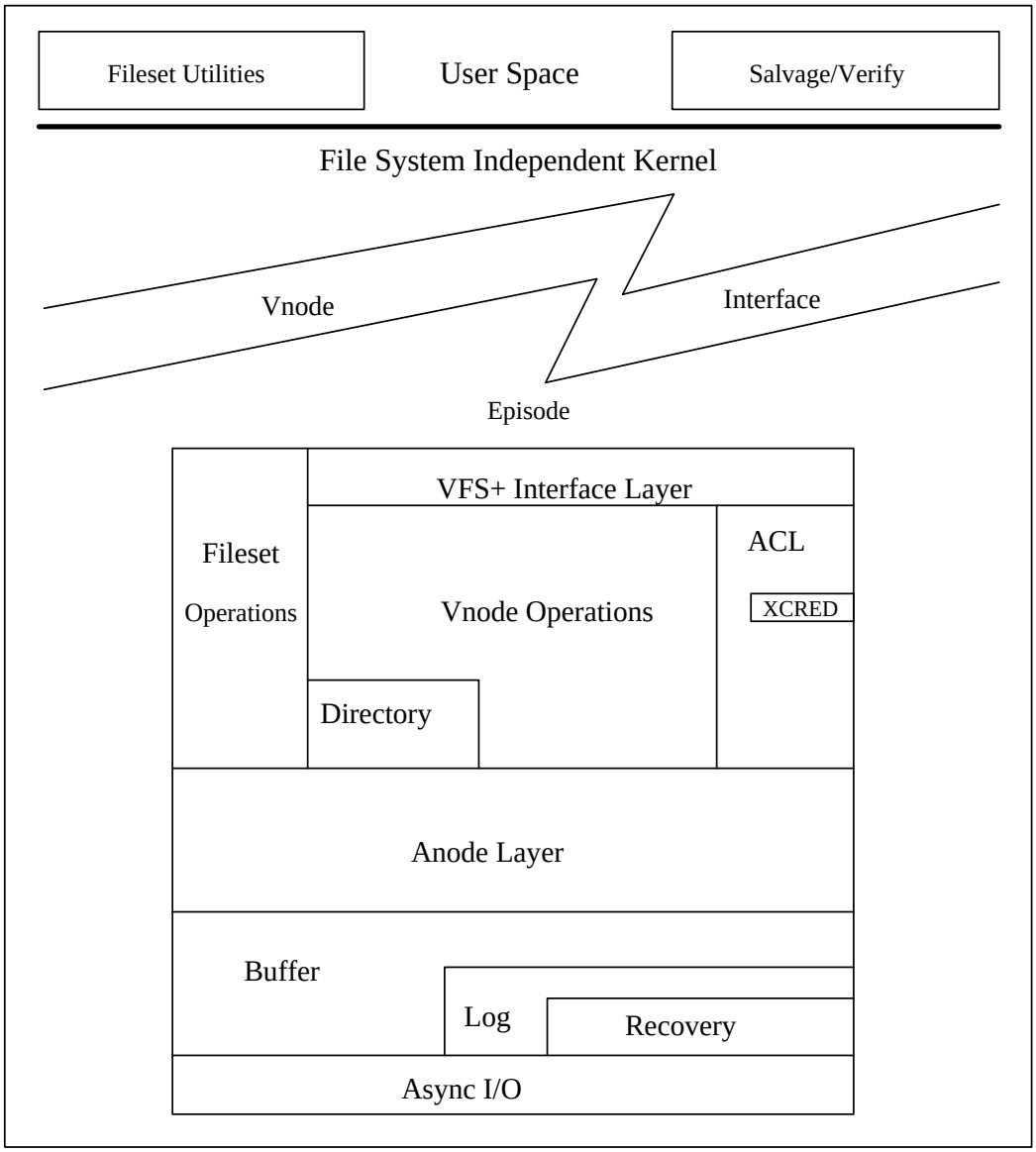


Figure 3: Layering in Episode.

can not regenerate a consistent file system, however, such as when hard I/O errors occur while updating critical meta-data. In such cases, the Episode *salvager* utility needs to be run; the salvager's performance characteristics are very similar to fsck.

The anode layer manages all references to data stored in containers. The container abstraction provides for three modes of storage: The *inline* mode uses extra space in the anode to store small quantities of data. This allows for efficient implementation of symbolic links, ACLs, and very small files. The *fragmented* mode enables several small containers, too big for inline storage, to share a disk block. Fragments are used for files smaller than a block. Finally, the *blocked* mode describes large containers. Four levels of indirect blocks [MCK 84] can be used to address $2^{31}$ block addresses. Due to other restrictions however, the maximum size of a file is bound by MIN($2^{32} * fragmentSize$, $2^{31} * blockSize$). Thus, if the fragment size is 1K, and the block size is 8K, a file can grow to $2^{42}$ bytes.[2] Block allocation policies try to ensure

[2] Additional kernel modifications, such as changes to the lseek system call interface, are required to use files of this size

contiguity, and support is provided for sparse files.

Directories are implemented straightforwardly as specially typed containers. Episode augments the directory implementation with hash tables to reduce search processing. Each 8K directory page contains its own separate hash table.

The Episode vnode layer extends the vnode operations designed by Sun Microsystems [KLE 86, ROS 90] with support for ACLs and filesets. In addition, the vnode operations that read or write files are integrated with the virtual memory system on SunOS 4.0.3c and AIX 3.1, allowing Episode to use the virtual memory pool as a file cache. This greatly improves the performance of the read and write operations on files, due to the increased cache size. Episode has also been optimized to detect sequential access and coalesce adjacent reads and writes.

## Logging Architecture

Typical transactional systems use *two-phase locking* (2PL) for ensuring consistency of data that is modified within a transaction. In two-phase locking, a transaction may, from time to time, obtain new locks, but it can never release any locks until the second "phase," when the transaction commits. By forbidding the release of locks until after the commit, this scheme guarantees that no other transactions ever read uncommitted data. Without two-phase locking rules, one transaction could lock, modify and commit data already modified and unlocked by a still-running transaction. An example is given below.

2PL ensures serializability and atomicity of the transactions, but at a cost: it reduces the concurrency in the system if the data being locked is a hot spot, since all the transactions that wish to obtain a lock on the hot spot must wait for the entire transaction currently holding the lock to complete.

In addition, using 2PL can add complexity to interface design in layered, modular systems. In a layered system, code in a higher layer typically calls code in a lower layer, which may lock its own private objects for the duration of a call. Quite often these locks are not exported. In order to use 2PL in such a model, one has to export details of the locks obtained by the lower level modules, since the locks they obtain remain locked until the high level transaction commits, and failure to set these low-level locks in the proper order could lead to deadlock. Two-phase locking thus greatly increases the complexity of such a layered interface.

To better understand the problem addressed by two-phase locking, which is known in the database literature as the problem of *cascading aborts*, consider the scenario in Figure 4, where two-phase locking is not performed:

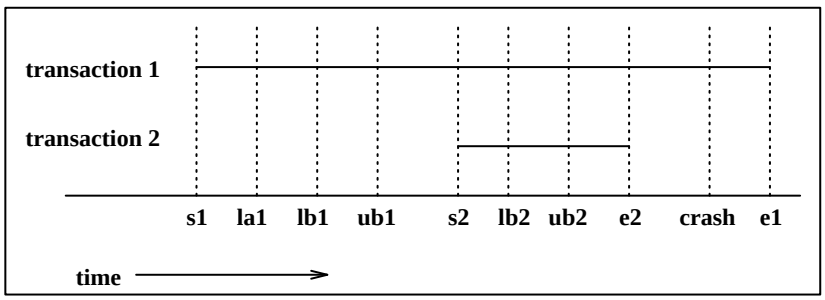


Figure 4: Formation of an Equivalent Class.

Transaction 1 starts at time **s1**, and locks two objects, **A** and **B**, at times **la1** and **lb1**, respectively. Transaction 1 shortly thereafter unlocks object **B** at time **ub1**. Next, before transaction 1 commits, transaction 2 modifies object B: transaction 2 starts at time **s2**, and locks object B at time **lb2** and finally

within most kernels.

unlocks object B at time **ub2**. Finally, transaction 2 commits this change to object B at time **e2**. Now, assume the system crashes shortly thereafter, at the time marked **crash**. After the log replay occurs, object **B** will contain the changes made by transactions 1 and 2, since transaction 2 committed these changes to this object. However, object **A** does not have the change made by transaction 1 committed, since transaction 1 never committed. The result is that only one of the changes made by transaction 1, the change to object B, is actually made permanent.

Episode's transaction manager avoids these problems by using a type-specific approach to transactional locking instead of two-phase locking. Episode transactions can acquire locks when they need them, and drop them as soon as they are finished using them, rather than waiting for the transaction's end. This allows for greater concurrency in the system, but required alternate mechanisms to prevent uncommitted data from being read by other transactions.

In order to avoid the problem of updating uncommitted data, Episode aborts transactions that might have otherwise been able to commit, if a crash intervenes. Specifically, all active transactions that lock the same object during their lifetime are merged into an *equivalence class* (EC). An EC has the property that either all of its transactions commit or none do.[3] In the example above, transaction 1 and 2 would form an EC. An EC can be viewed as an umbrella transaction that subsumes all the transactions that belong to it. ECs are formed whenever active transactions exhibit read-write sharing amongst themselves.

If the system crashes before all of the transactions in an EC have committed, all the transactions in the EC are aborted. It is therefore desirable to minimize the duration and the number of ECs formed in the system. To this end, transactions typically delay the use of "hot" data until as close to the transaction's end as possible to minimize the chance that some other transaction will have to read this data before it commits.

The primary goal of the logging system in Episode is to guarantee the consistency of the file system. This decision impacted a number of other design choices:

Meta-data changes to the disk itself can often be deferred, unless specifically requested by an operation like *fsync*. Consistency of the file system doesn't require that the system be current.

No transactional guarantees are required about the user-data, since consistency of the file system requires only that the meta-data be consistent. Episode logs only meta-data changes. Although restricting logging to meta-data greatly reduces log traffic, mixing logged and unlogged blocks in the same file system introduces some complexity.

To illustrate some of these issues, note that if a crash occurs between the time that a data block is allocated to a file and the time that the block is first written, the former data may appear in the new file as uninitialized data. The problem arises because the allocation update commits transactionally while the data update fails, since the data update occurs outside of the transaction system. Episode fixes this problem by starting another transaction when the block is allocated, and ending it when the first write to it completes. If the system crashes and the transaction aborts, the recovery procedure for this special transaction zeroes the contents of the block.

Since users do not define the start and end times of transaction, transaction sizes can be bounded when they begin. This allows the use of a very simple algorithm to ensure that running transactions never exceed the space available in the log. Transactions that run too long or modify too much data represent programming errors.

As mentioned above, Episode logs both the new value and the old value of the data being modified. A number of other systems simply log the new value of the modified data. In systems that log only new data, dirty data can not be written out to its final home on the disk until the transaction actually commits, since the log does not contain enough information to undo the updates, and the transaction manager can

[3] Each transaction by itself forms an equivalence class with one member.

not redo the updates to get to a consistent state until the transaction has ended and thus made all of its modifications. Our design choice was significantly influenced by our concern that using new value-only logging would seriously constrain the buffer package's choice of which buffers to write out to the disk, and when to write these buffers. Old value / new value logging, on the other hand de-couples the writing of buffers from the end of transactions, at the expense of having to write more data to the log.

Introducing logging in Episode affected the implementation of all the vnode operations. The bound on the transaction size required by our log space allocation policy dictated that complicated and time consuming operations be broken up into smaller bounded operations, each of which can be bracketed transactionally. For example, a delete of a large file is broken into a sequence of separate file truncation operations, followed by a deletion of the empty file. After each transaction, the file system is consistent, if not in the final desired state.

## Performance

This section details the result of running benchmarks that measure the performance of Episode and a reference file system (The IBM RS/6000's JFS or Sun's BSD file system), both on meta-data and I/O operations. Comparison with Sun's BSD illustrates the effects of logging in Episode, while comparison with JFS, which also uses logging, measures the efficiency of our implementation. All measurements were taken on the following platforms:

- A SUN SPARCstation 1 running SunOS 4.0.3c, with 8MB memory and 200MB Seagate ST1239NS SCSI disk (peak data transfer rate 3.0 Mbytes/sec average latency 8.33 msec).
- An IBM RS/6000 Model 320 running AIX 3.1, with 32MB of memory and 320MB IBM 0661-371 SCSI disk (peak data transfer rate 2.0 Mbytes/sec, average latency 7.0 msec).

Our performance goals were that Episode perform meta-data update operations significantly faster than the Berkeley Fast File system, while doing large I/O operations essentially as fast as the native disk driver would perform large transfers. We expect that our meta-data update operations would be considerably faster than BSD's because Episode batches meta-data updates into writes to the file system log.

In terms of the experiments done in this section, we thus would hope to do meta-data update operations considerably faster than the SunOS BSD implementation, and normal read and write operations essentially as fast as the JFS implementation.

One would also expect that Episode would perform I/O somewhat faster than the SunOS 4.0.3 BSD implementation. However, our integration of Episode with the SunOS virtual memory system is not yet complete. In particular, on that platform, Episode does no read-ahead, nor are any write operations asynchronous, and these problems significantly impact the SunOS read and write performance figures. Under AIX 3.1, we have completed this level of virtual memory system integration, and on that platform, we expected that our performance would be essentially as good as IBM's JFS. As an aside, the problems in doing read-ahead and asynchronous I/O in AIX 3.1 and SunOS 4.X are quite similar, and we do not expect any serious problems in completing the SunOS implementation.

In the next sections, we present the results of various performance tests, and some discussion on the results.

## Connectathon Benchmark

The *connectathon* test suite tests the functionality of a UNIX-like file system by exercising all the file system related system calls. It consists of nine tests:

- *Test1* creates a multi-level directory tree and populates it with files. A meta-data update intensive test.
- *Test2* removes the directories and files created by test1. A meta-data update intensive test.
- *Test3* does a sequence of *getwd* and *stat* on the same directory. Primarily meta-data read operations.
- *Test4* executes a sequence of *chmod* and *stat*, on a group of files. A meta-data update intensive test.
- *Test5* writes a 1 MB file sequentially, and then reads it. Primarily I/O operations.
- *Test6* reads entries in a directory. Primarily meta-data read operations.
- *Test7* calls *rename* and *link* on a group of files. A meta-data update intensive test.
- *Test8* creates symbolic links and reads them by *symlink* and *readlink* calls respectively, on multiple files. Primarily a meta-data update intensive test, with significant meta-data reading, too.
- *Test9* calls *statfs*.

Figures 5 and 6 compare Episode performance with JFS on an RS/6000, and with BSD on a Sparcstation.[4]

The most interesting numbers in this section come from a comparison of Episode and BSD on the Sun platform. Those tests representing primarily meta-data updates (tests 1, 2, 4, 7 and 8) show the benefits of logging on meta-data updates; in all but one test, Episode does at least twice as well as BSD in elapsed time. The test that gives Episode difficulty, test2, does a lot of directory I/O operations. These operations use a private buffer cache, one that appears from our examination of read and write counts to be too small.

In addition, we compared Episode with another log-based file system, JFS. This was done as an additional check on our implementation, to verify that our performance was approaching that of a highly tuned commercial file system with a somewhat similar architecture. These figures show that Episode performance on meta-data operations is comparable or better than that of JFS in terms of elapsed time.

In addition to comparing Episode in elapsed time, we also measured the CPU utilization in Figures 7 and 8. In both of these sets of figures, Episode's CPU utilization is higher than that of the native file system. We will discuss reasons for this below, but we should point out that we expect this situation to improve as Episode's performance is tuned further. In particular, for the meta-data reading tests, Episode is CPU-bound, and we expect further reductions in CPU usage to map directly to reductions in elapsed time.

| | | Meta-data updates | | | | | Other | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | test1 | test2 | test4 | test7 | test8 | test3 | test5 | test6 | test9 |
| *Elapsed* | *Episode* | 4 | 3 | 2 | 1 | 2 | 10 | 2 | 4 | 0 |
| *Time* | *JFS* | 6 | 3 | 1 | 5 | 6 | 3 | 10 | 8 | 0 |

Figure 5: Comparison of Episode with JFS on the RS/6000 platform, executing the Connectathon tests. The numbers listed are averages of several runs. All figures are elapsed times in seconds.

## Other Benchmarks

We also ran two tests representing a mix of file system operations: the modified Andrew benchmark [OUS 90, HOW 88], and the NHFSTONE benchmark from Legato Systems (v1.14).

[4] In some instances, elapsed time appears to be less than the CPU time, due to difference in the granularity of measurement.

| | | Meta-data updates | | | | | Other | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | test1 | test2 | test4 | test7 | test8 | test3 | test5 | test6 | test9 |
| *Elapsed* | *Episode* | 7 | 6 | 4 | 2 | 4 | 0 | 46 | 6 | 1 |
| *Time* | *SunOS* | 15 | 6 | 17 | 11 | 13 | 0 | 25 | 14 | 0 |

Figure 6: Comparison of Episode with BSD on the Sun platform, executing the Connectathon tests. The numbers listed are the averages of several runs. All figures are elapsed times in seconds.

| | | Meta-data updates | | | | | Other | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | test1 | test2 | test4 | test7 | test8 | test3 | test5 | test6 | test9 |
| *CPU* | *Episode* | 1.7 | 1.2 | 2.6 | 1.7 | 2.0 | 10.0 | 1.8 | 4.0 | 0.6 |
| *Time* | *JFS* | 0.6 | 0.5 | 1.0 | 0.8 | 0.9 | 2.9 | 1.9 | 1.4 | 0.3 |

Figure 7: Comparison of Episode with JFS on the RS/6000 platform, executing the Connectathon tests. The numbers listed are averages of several runs. All figures are CPU times in seconds.

The modified Andrew benchmark was originally devised to measure the performance of a distributed file system, and operates in a series of phases, as follows:

The benchmark begins by creating a directory tree, and copying the source code for a program in that tree. It then performs a *stat* operation on every file in the hierarchy. It subsequently reads every file as part of compiling them, using a modified GNU C compiler that generates code for an experimental machine.

The results of running this benchmark on the RS/6000 configuration above were that JFS completed the test in 90.0 seconds, while Episode took 102.2 seconds. Most of the difference between the two tests occurred on the stat and copy phases of benchmark.

The NHFSTONE benchmark from Legato Systems, Inc. (v1.14) was altered to be a local file system benchmark instead of an NFS benchmark. The dependence on kernel NFS statistics was removed and the benchmark was run locally on the server rather than over the network from a client that has mounted an NFS exported file system. The standard mix of operations was used to test the throughput of JFS and Episode, i.e., 13% fstat, 22% read, 15% write, etc. The tests were run on a 32MB IBM RS/6000 running AIX 3.1 release 3003. A Fujitsu M2263 disk was used to hold the file systems in the tests.

The test results indicate that JFS reaches a peak throughput level of about 233 file system operations per second (with 2 processes) while Episode reaches about 300 operations per second (with about 10 processes). In doing so, Episode used roughly twice as much CPU per operation as JFS to achieve these higher throughput levels.

In short, Episode ran slightly slower than JFS on the modified Andrew benchmark, and slightly faster than JFS on the NHFSTONE benchmark. We feel that the performance of Episode on these benchmarks is quite acceptable, given the tuning that will be done as vendors ship Episode.

## Read and Write

Episode's ability to utilize the available disk bandwidth is shown in the comparison with JFS on the RS/6000 on the read and write tests. Two types of tests were run, one type measuring cached read performance, and one type measuring uncached read and write performance.

Both the cold cache read performance numbers (Figure 9) and the cold cache write performance numbers (Figure 11) show quite similar performance between JFS and Episode. We believe that this indicates that Episode's algorithms for doing I/O operations in large chunks are working reasonably well.

The Episode warm cache read rate is a bit slower than the corresponding JFS rate, as can be seen in

| | | Meta-data updates | | | | | Other | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | test1 | test2 | test4 | test7 | test8 | test3 | test5 | test6 | test9 |
| *CPU* | *Episode* | 2.8 | 2.1 | 3.8 | 2.0 | 4.0 | 0.7 | 15.4 | 5.1 | 0.9 |
| *Time* | *SunOS* | 1.3 | 0.8 | 2.0 | 0.3 | 1.9 | 0.3 | 6.1 | 1.5 | 0.4 |

Figure 8: Comparison of Episode with BSD on the Sun platform, executing the Connectathon tests. The numbers listed are the averages of several runs. All figures are CPU times in seconds.

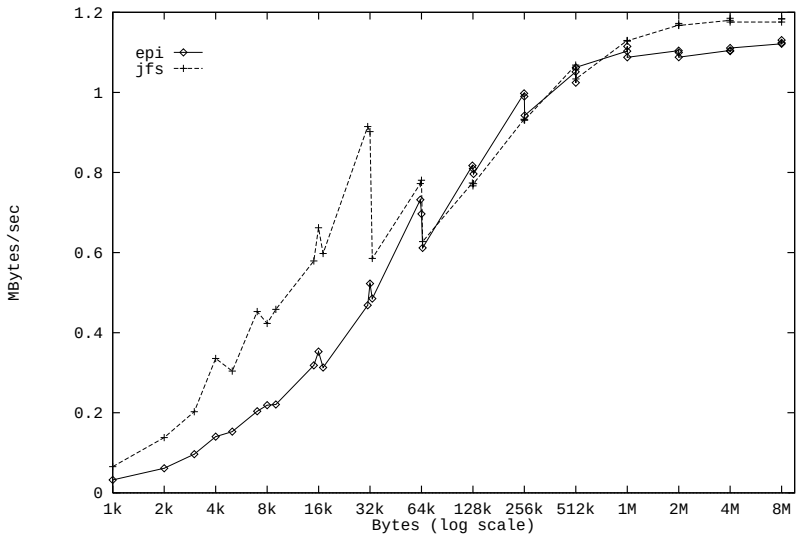


Figure 9: Comparison of Episode and JFS Read Rates - Cold VM Cache.

Figure 10. This rate measures how quickly the file system can locate its data and copy it, or map it, from the virtual memory system into the caller's buffers. As such, it is not as much of a file system performance tests as a virtual memory integration performance test. These figures peak between 16 and 20 megabytes per second, well above the disk's actual data transfer rate.

It is clear from comparing the warm and cold read performance numbers that the key to good system performance is successful integration with the virtual memory system.

## Performance Summary

Episode performs well in handling basic read and write operations, doing I/O in as large a chunk as useful. In this area of our design, we borrowed heavily from the work done on both AIX's JFS and SunOS's BSD file systems [MCV 91] on obtaining extent-like performance from BSD-style file system organizations.

Episode's greatest performance benefits come in its performance on meta-data operations. In these operations, the use of logging greatly reduces the number of synchronous write operations required, significantly improving system performance.

In addition, Episode is a relatively new file system, and is still undergoing significant performance measurement, profiling and tuning. We used the tracing and profiling facility on the RS/6000 to produce traces that recorded each procedure entry and exit along with timing information. A detailed study of the these traces on micro-benchmarks identified a wealth of targets for optimization. In particular we found that:

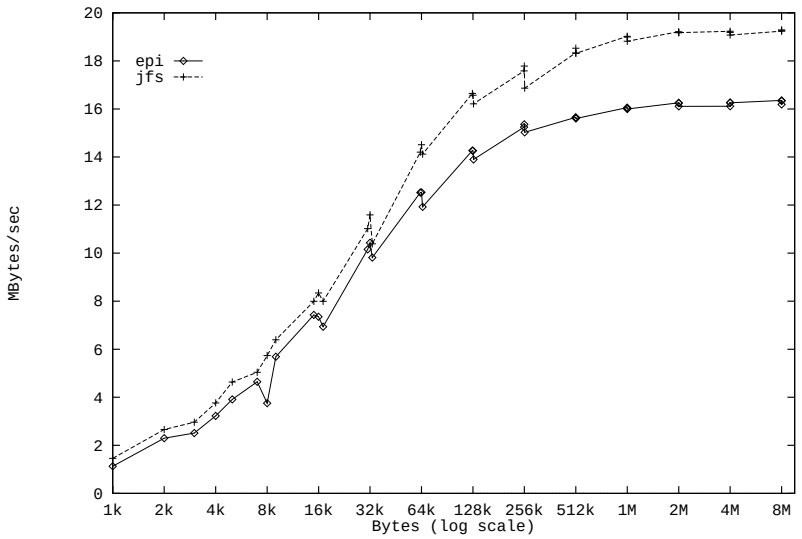


Figure 10: Comparison of Episode and JFS Read Rates - Warm VM Cache.

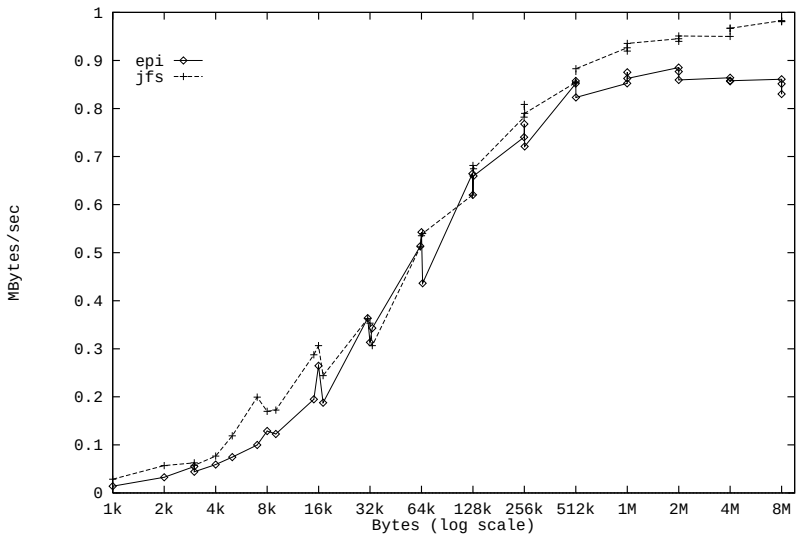


Figure 11: Comparison of Episode and JFS Write Rates.

- Episode is not passing enough context information between layers,
- certain invariant computations are being performed repeatedly,
- common data paths are using expensive general-purpose interfaces, where a special case data path would be more efficient, and
- various parameters, like the size of in-core caches for the vnodes, anodes and buffers, have not been tuned.

We expect CPU usage to drop considerably as we further optimize the code.

The integration of Episode with the virtual memory system under SunOS is still incomplete; in particular, read-ahead and asynchronous write are not yet implemented on that platform. As a result, the performance of Episode on the Sun, using test5, is relatively poor as compared with BSD. On the RS/6000 Episode is better integrated with the native virtual memory system, does perform read-ahead and asynchronous writes, and exhibits read-write performance comparable with the local file system, JFS. This leads us to expect that the implementation on the Sun will perform equally well, once the Sun port is completed.

## Recovery Time

Episode's time to recover depends primarily on the size of the active portion of transaction log. The active portion of the log is that part of the log that needs to be replayed after a crash, and must include all of the uncommitted transactions, since these must be undone in the event of a crash. The active portion of the log may go back even further, should the buffer cache still contain dirty meta-data blocks that were modified by committed transactions. In this case, the updates are in the log and only in the log, requiring the replay of that part of the log in the event of a crash. The operation of writing buffers modified by committed transactions and discarding those portions no longer required to ensure the permanence of those transactions is generally called *checkpointing* the log.

In order to estimate the size of the active portion of the log after a crash, note first that no matter how often the system is checkpointed, there is no way to avoid an active portion of the log containing at least those transactions that are currently executing. Thus, as system activity at the time of a crash increases, we should see the minimum recovery time rise correspondingly. In addition, if the log is checkpointed only every T seconds, as is the case with Episode, then the active portion of the log can rise to include all the transactions that modified the dirty buffers still resident in the buffer cache. Of course, Episode will not permit the log to become full, but it is difficult to guarantee any other bound on the size of the active portion of the log.

Of course, the time to replay a block of the active log is not constant, but is bounded: There is a maximum number of meta-data blocks whose updates can be described by a block of the log, but many log blocks will effect considerably fewer meta-data blocks.

From the above discussion, the reader can see that the recovery time for an Episode partition should rise in proportion to the number of processes actively modifying the file system at the time of the crash, but that there will be a number of recovery calls that take somewhat longer than the minimum, because of uncheckpointed, but committed, transactions.

In order to verify this state of affairs, we ran some experiments, timing recovery on a 900 megabyte aggregate, with a 9 megabyte log, in two configurations: with 6 megabytes of new data stored in the aggregate, and with 260 megabytes of new data stored in the aggregate.

After crashing the system with 10 active processes modifying the 6 MB file, recovery took between 4.1 and 6.7 seconds to execute, while after crashing the system with 20 active processes modifying the same 6 MB file system, recovery took 10.5 seconds on the single instance we ran. Similarly, in an experiment on the 260 MB file system, crashing the system with 10 active processes took between 5.1 and 8.8 seconds to recover, while crashing the system with 20 active processes took 2.7 seconds to recover on the single instance we ran. From this data, we can see that recovery takes essentially the same amount of time on small and large aggregates.

On the other hand, there was a noticible, if highly varying, correspondence between system activity at the time of a crash, and recovery time. In tests with the 6 megabyte file system, recovering after a crash with one active process took between 1.7 and 2.7 seconds. Recovering after a crash with 5 processes took between 1.9 and 9.2 seconds. Recovering with 10 processes took between 4.1 and 6.7 seconds. Recovering with 20 processes took 10.5 seconds (one data point), and recovering after a crash with 49 active processes took 19.6 seconds.

In conclusion, we note that the time to recover depends in a complex way upon a number of variables, none of which, however, are the aggregate size. Despite this complexity, it also appears that in typical configurations, recovery times will be under 30 seconds.

## Status

Episode is functionally complete, and is undergoing extensive stress testing and performance analysis. Episode will ship with the DCE as the Local File System (LFS) component, and also works with Sun's Network File System [SAN 85]. Episode is designed to be portable to many kernels, and presently runs on SunOS 4.0.3c, SunOS 4.1.1 and AIX 3.1.

The design of Episode began in 1989, and full-scale implementation began in January 1990. The file system was first tested in user space and then plugged into the kernel, saving considerable amounts of debugging in the kernel environment. The present code, which includes substantial debugging code, test suites, scaffolding to run tests in user space, and utilities, is about 70K lines of C, according to "wc".

## Conclusions

The abstraction of containers has proved to be very useful. By separating the policy from the mechanism for placing the data on the disk, the container abstraction helps isolate the code responsible for data location and allocation, as well as making many structures extensible "for free." The resulting flexibility in data layout policies enables future releases of Episode to use more knowledge in allocating space for user data and meta-data, while leaving the disk format itself formally unchanged.

Our experience with Episode also shows that a general purpose transactional system is not required for a file system. The Episode log implements only a small subset of the functionality needed in a database system, and our log and recovery packages are but a fraction of the size of those in traditional database products.

On the other hand, the Episode transaction manager must deal with a few technicalities not present in most database systems. There are some complications introduced by a design storing both meta-data and unlogged user data on the same disk. Furthermore, the decision to form equivalence classes of transactions instead of using two-phase locking also required new code.

The original motivation for implementing a log-based system was fast crash recovery, but we are obtaining substantial performance benefits as well. Logging has improved the performance of meta-data updating operations and reduced the cumulative disk traffic by permitting Episode to batch repetitive updates to meta-data.

The key to obtaining good performance of read and write operations was a successful integration with the virtual memory system, and performing I/O in large blocks. We confirmed the results that disk bandwidth is utilized more efficiently when data transfers occur in large chunks. The virtual memory system provides a very effective memory cache for files, and also enables the merging of requests for adjacent disk blocks into one large request. In general, however, virtual memory systems exhibit a great deal of idiosyncratic behavior, and are sufficiently diverse that the integration process is very difficult.

## Acknowledgements

We are grateful to Alfred Spector for comments and corrections. Thanks go to Mike Comer, Jeffrey Prem, Peter Oleinick and Phil Hirsch for running some of the benchmarks, and the POSIX compliance tests.

We would also like to thank various people in IBM for discussing various file system performance issues with us, including Al Chang, Carl Burnett, Bryan Harold, Liz Hughes, Jack O'Quin, and Amal

Shaheen-Gouda.

## References


CHA 88 A. Chang, and M. F. Mergen. 801 Storage: Architecture and Programming. *ACM Trans. Computer Systems* 6, February 1988.

CHA 90 A. Chang, M. F. Mergen, R. K. Rader, J. A. Roberts, and S. L. Porter. Evolution of storage facilities in AIX Version 3 for RISC System/6000 processors. *IBM Journal of Research and Development*, Vol. 34, No. 1, January 1990.

GIN 87 Robert A. Gingell, Joseph P. Moran, and William A. Shannon. Virtual memory architecture in SunOS. *Usenix Conference Proceedings*, Summer 1987.

HAE 83 T. Haerder, A. Reuter. Principles of Transaction-Oriented Database Recovery. *Computing Surveys*, Vol. 15, No. 4, December 1983.

HAG 87 Robert B. Hagmann. Reimplementing the Cedar File System Using Logging and Group Commit. *Proceedings of the 11th Symposium on Operating Systems Principles*, November 1987.

HOW 88 J. H. Howard, M. L. Kazar, S. G. Nichols, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, Vol. 6, No. 1, February 1988.

KAZ 90 Kazar, Leverett et al. DEcorum File System Architectural Overview. *Usenix Conference Proceedings*, June 1990.

KLE 86 S. R. Kleiman. Vnodes: an Architecture for Multiple File System Types in Sun UNIX. *Usenix Conference Proceedings*, Summer 86.

KOW 78 T. Kowalski. *FSCK: the UNIX system check program.* Bell laboratory, Murray Hill, NJ 07974. March 1978.

LEF 89 S. Leffler, M. McKusick, M. Karels, and J. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley, 1989.

MCK 84 McKusick, M.K., W.N. Joy, S.J. Leffler, R.S. Fabry. A Fast File System for UNIX. *Transactions on Computer Systems*, Volume 2, No. 3, August 1984.

MCK 90 M. McKusick, M. J. Karels, and Keith Bostic. A Pageable Memory based File System. *Usenix Conference Proceedings*, Summer 1990.

MCV 91 L. W. McVoy, and S. R. Kleiman. Extent-like Performance from a UNIX File System. *Usenix Conference Proceedings*, Winter 91.

MOG 83 Jeffrey Mogul. *Representing Information about Files.* Computer science department, Stanford university, CA 94305. September 1983.

MOH 89 C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, P. Schwarz. *ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks using Write-ahead Logging.* Research report, IBM research division, Almaden Research Center, San Jose, CA 95210. January 1989.

OUS 90 John K. Ousterhout. Why aren't Operating Systems getting faster as fast as Hardware ? *Usenix Conference Proceedings*, June 1990.

PEA 88 J. K. Peacock. The Counterpoint Fast File System. *Usenix Conference Proceedings*, Winter 1988.

RED 89 A. L. Narasimha Reddy, and P. Banerjee. An Evaluation of Multiple-Disk I/O Systems. *IEEE Transactions on Computers*, Vol. 38, No. 12, December 1989.

REN 89 R. Van Renesse, A. S. Tannenbaum, and A. Wilschut. The Design of a High-Performance File Server. *Proc. Ninth Int'l Conf. on Distributed Comp. Systems*, IEEE, 1989.

ROS 90 Mendel Rosenblum, John K. Ousterhout. The LFS Storage Manager. *Usenix Conference Proceedings*, June 1990.

ROSD 90 David S.H. Rosenthal. Evolving the Vnode Interface. *Usenix Conference Proceedings*, Summer 1990.

SAN 85 R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network File System. *Usenix Conference Proceedings*, Summer 1985.

SAT 85 M. Satyanarayanan, J. H. Howard, D. A. Nichols, R. N. Sidebotham, and A. Z. Spector. The ITC Distributed File System: Principles and Design. *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, 1985.

SID 86 R. N. Sidebotham. Volumes: The Andrew file system data structuring primitive. *European Unix User Group Conference Proceedings*, August 86.

STA 91 C. Staelin, and H. Garcia-Molina. Smart File Systems. *Usenix Conference Proceedings*, Winter 1991.

VER 91 Veritas Software Corporation. *VERITAS Overview* (slides). Veritas Software, 4800 Great America Parkway, Suite 420, Santa Clara, CA 95054.

## Biographical Information

**Owen T. Anderson** is a member of the File Systems Development group. He worked on file system security while a member of the Andrew File System group at Carnegie Mellon University's Information Technology Center. At Transarc, he continues this specialization and also contributes to design efforts and kernel development. Before coming to Carnegie Mellon, Mr. Anderson worked for ten years at the Lawrence Livermore National Laboratory in Livermore, California. There he obtained a wide variety of experience ranging from the design of an operating system and two multi-processor architectures to debugging digital hardware. Owen Anderson graduated from the Massachusetts Institute of Technology in 1979 with an S.B. degree in Physics. He can be reached via e-mail at ota@transarc.com.

**Sailesh Chutani** has been involved with the Andrew File System (AFS) project since June 1988 when he joined Carnegie Mellon University's Information Technology Center. At Transarc, he was one of the designers of AFS 4. He continues work on the design and development of AFS. Mr. Chutani holds an M.S. in Computer Science from the University of North Carolina at Chapel Hill and a B.Tech. in Computer Science and Engineering from the Indian Institute of Technology at Kanpur, India. He can be reached via e-mail at chutani@transarc.com.

In his role as Manager of File Systems Architecture, **Dr. Michael L. Kazar**, one of Transarc's founders, has full responsibility for the development of Transarc's distributed file systems products. This undertaking is a natural combination of Dr. Kazar's previous work as "Senior Data Stylist" at Carnegie Mellon University's Information Technology Center. In that position since 1984, he was instrumental in the design and implementation of the Andrew File System, assuming responsibility for the management of that project in early 1988. While at the ITC, Dr. Kazar also worked on other aspects of file systems and on user interfaces. Dr. Kazar received two S.B. degrees from the Massachusetts Institute of Technology, and his Ph.D. in Computer Science from Carnegie Mellon University, in the area of optimizing multiprocessor computations to minimize communications costs. He can be reached via e-mail at kazar@transarc.com.

Prior to joining Transarc, **Dr. Bruce W. Leverett** worked for seven years at Scribe Systems (formerly Unilogic). There he participated in development of the Scribe document production system,

including the Scribe text formatter and an X-Windows-based PostScript Previewer. He developed source-to-source program translation technology to enable Scribe software to be ported to multiple platforms. Dr. Leverett completed his doctoral dissertation at Carnegie Mellon in 1980. His thesis research, in optimizing compilers, was an outgrowth of previous work in that field, including development of compilers for the BLISS language, and research in language design and implementation for multiprocessors, including implementation of a variant of Algol 68 for the Hydra operating system. He holds an A.B. from Harvard in Physics and Chemistry, completed in 1973. While at Harvard, he implemented a chess-playing program, which competed in the ACM Computer Chess Championship in 1972. He can be reached via e-mail at bwl@transarc.com.

**W. Anthony Mason** is a member of the AFS team and specializes in data communications. Prior to joining Transarc, Mr. Mason served as a Systems Programmer in the Distributed Systems Group at Stanford University in the Department of Computer Science. He was involved in the development of both the V distributed system and the VMTP transport protocol. Mr. Mason received his B.S. degree in Mathematics from the University of Chicago. He can be reached via e-mail at mason@transarc.com.

**Robert N. Sidebotham** was a key designer of the Andrew File System at the Information Technology Center of Carnegie Mellon University, and the inventor of *volumes* (now *filesets*), which pervade the design, implementation, and administration of AFS and its descendents. Bob has been involved in a variety of other software projects, from digitizing and imaging of satellite data from the Canadian satellite, ISIS-II, to the rendering of architectural drawings, to the design and implementation of an operating system for Sweden's Teletex system. He is also a founder of a Pittsburgh-based niche software company, which he left in 1991 to join Transarc. Mr. Sidebotham graduated from the University of Calgary, Alberta, Canada, in 1974, with a BSc. in Computing Science. He can be reached via e-mail at bob@transarc.com.

## Availability

Episode is designed to be portable to many kernels, and presently runs on SunOS 4.0.3c, SunOS 4.1.1 and AIX 3.1. It is available as the Local File System component of the OSF's Distributed Computing Environment, and is also licensable as a separate standalone product from Transarc Corporation.