# Reimplementing the Cedar File System Using Logging and Group Commit

Robert Hagmann
*Xerox PARC*

**Abstract:** The workstation file system for the Cedar programming environment was modified to improve its robustness and performance. Previously, the file system used hardware-provided labels on disk blocks to increase robustness against hardware and software errors. The new system does not require hardware disk labels, yet is more robust than the old system. Recovery is rapid after a crash. The performance of operations on file system metadata, e.g., file creation or open, is greatly improved.

The new file system has two features that make it atypical. The system uses a log, as do most database systems, to recover metadata about the file system. To gain performance, it uses group commit, a concept derived from high performance database systems. The design of the system used a simple, yet detailed and accurate, analytical model to choose between several design alternatives in order to provide good disk performance.

## 1. Introduction

Workstation file systems ought to provide good performance, rapid crash recovery, and robustness. This paper will discuss four implementation approaches used to achieve these goals in the reimplemented Cedar workstation file system:

1. log-based recovery for metadata, such as the file name table,
2. use of group commit to avoid some disk writes,
3. a design based on a simple performance model that captures the timing characteristics of disks, and
4. techniques (e.g., writing key system data in two places) to increase the robustness of the system. These techniques are well known, but they were implemented at little performance cost because of the use of the log and group commit.

PARC has a history of building file systems that are quite robust. Cedar [Swin86] is the programming environment in use at Xerox PARC/CSL. To be accepted by the Cedar users, the new file system had to be at least as robust as the existing system. The new file system is part of the Cedar release of March 1987, which runs on present hardware.

File systems have been studied extensively. For example, McKusick et al. reported on their improvements to the performance of the UNIX[1] file system [McKu84]. The work reported here has been done in the tradition of the Xerox workstation file systems: the Alto file system [Lamp79a], the Pilot file system [Rede80], and the Cedar file system [Schr85].

What distinguishes this work from other file systems is that it incorporates some ideas from database systems. Logging is commonly used in these systems; see [Gray79] for a discussion of logging. Group commit is part of the lore of database systems; a reference to it is in [DeWi84].

## 2. History, Terminology and Background

Cedar currently runs on the "D-machine" architecture, exemplified by the Dorado [Lamp81] and Dandelion computers. These D-machines use the Trident disk interface that permits an implementation of "labels" for every disk sector (see below for a discussion of labels). A new VLSI multiprocessor workstation, the Dragon [McCr84], is under development and will conform to some standard I/O interface that will not support labels. Thus, to port the Cedar operating system to the Dragon, the existing file system required modification. This provided an

[1] UNIX is a registered trademark of AT&T Bell Laboratories.

opportunity to improve the performance and robustness of the file system.

One of the features of the Alto [Thac79] and its successors has been their *robust* file system. A robust system is one that continues to operate (nearly) correctly in the presence of some class of errors. These file systems are robust against single page disk errors and memory smashes.

In the Trident disk interface, each sector has an additional field, the *label field*, usually not found in other interfaces. PARC file systems for these disks use the label to mark each sector with information identifying the sector. In normal operation, before a sector's data is read or written, the label is verified against a label computed by the file system. If revolutions are not to be dropped, this check must be performed in microcode. The addition of the label field makes it possible to detect, in hardware and microcode, errors such as wild writes. File allocation, extension, contraction, and deletion operations write the labels. Also, it is possible to *scavenge* the file system; by reading the labels and interpreting some of the disk sectors, file system structural information, such as the free page map and the file name table, can be reconstructed.

The Cedar File Package and File System, *FS*, together implement the abstraction of a named file [Schr85]. A file is a sequence of pages, logically numbered beginning with zero. The File Package is concerned with allocating and deleting files, opening them, reading and writing file pages, and keeping track of free pages. It specifically does not provide file names. This function is performed by FS. FS keeps the names of all the files in its file name table, which might be called a directory or catalog in other systems. For the rest of this paper we will call the previous version of the file system *CFS* (as in [Schr85]), and call the reimplementation *FSD* (FS for Dragon).

In CFS, a file has two kinds of sectors: *header* sectors and *data* sectors. Header sectors contain *file properties* (e.g., the file's name, length and create date) and a *run table* describing the extents of the file. The header sectors serve about the same purpose as the inodes do in the UNIX file system [Bach86], but have a different implementation. *Data sectors* store the contents of the file.

The Cedar File Package keeps a bit vector as a hint for which disk pages are free. This is called the Volume Allocation Map (VAM). There are no invariants associated with the VAM. Free pages may be lost and file creation may be somewhat slow, but these are judged to be acceptable performance problems and do not affect the integrity of the system.

## 3. Requirements for FSD

Five prime requirements for the file system are relevant here. Other requirements that are more system specific will not be covered in this paper.

First, the system should be robust against a hardware sector error. An error on *any* sector on the disk should only affect the file that contains that sector. The failure model allows for one or two consecutive sectors to fail, but longer contiguous failures are considered to be a massive failure from which complete recovery may not be possible. A flaw on the disk or a failed write usually damages at most one sector, but occasionally two contiguous sectors are damaged. More stringent requirements (e.g., loss of a whole track) can be met within the framework of the design presented below, but it would complicate the algorithms and have a minor performance impact. Loss of any part of the file name table should never occur due to a single sector failure. Massive failures (e.g., a head crash) are non-recoverable, except from backup. Mirrored hardware could be used to guard against massive failures [Lamp79b].

Second, the file system should be high performance. It must be designed so that the normal cases of read, write, create, and delete operate near "hardware speed."

Third, it should be robust against some software errors. FSD protects itself against two types of errors: memory smashes produced by other software and some classes of internal bugs.

Fourth, it should have fast recovery. Loss of a sector should not force a time consuming operation such as scavenge, which scans the disk and rebuilds the file name table. This is particularly critical as disk capacity continues to grow.

Finally, the system should use commercially available disk hardware. A new, label-free design is required.

## 4. Design Overview

CFS and FSD differ in the location and contents of their disk resident data structures. Table 1 shows a slightly simplified diagram of the location of file system metadata.

| **CFS** | **FSD** |
|---|---|
| File Name Table | File Name Table |
| text name | text name |
| version | version |
| keep | keep |
| uid | uid |
| header page 0 disk address | run table |
| | byte size |
| Headers | create time |
| run table | |
| byte size | Leaders |
| keep | uid |
| create time | preamble of run table |
| version | checksum of run table |
| text name | |
| | |
| Labels | |
| uid | |
| page number | |
| page type (header, free, data) | |

**Table 1. Disk Data Structures for Local Files in CFS and FSD**

There are three types of file name table entries in CFS and FSD: local files, symbolic links to remote files, and cached copies of remote files. The table only compares local files. This is sufficient to give a sense of the differences and reduces complexity in the table.

CFS splits the naming, property, and other information about files between three disk entities: the file name table, header sectors for files, and labels for every disk sector. To open a file from its name, the name is looked up in the file name table. The unique identifier (uid) and the "header page 0 disk address" are used to compute a label for the header of the file. The page at the disk address is read to get the file header. This gives CFS the run table and all the other properties for the file. A bug in the file system will often show up as an error in comparing the computed label with the disk label.

Note that most of the data is replicated or can be recomputed. For example, the "text name" of a file is stored both in the file name table and the header for the file. Also, the run table can be recomputed by reading all of the labels.

CFS creates a one byte file by finding three free pages from the VAM, reading their labels to check that they are really free, writing the labels to claim them for the file, writing the header, updating the file name table, writing the byte, and rewriting the header. Note that this is (at least) six I/O's.

FSD moves all the header information from the file headers directly into the file name table. While CFS kept most critical information in two places, FSD keeps all its information in the file name table. This improves the locality of the file system. Redundancy is achieved in FSD by keeping two copies of the file name table; in CFS redundancy was achieved by keeping different data structures.

FSD also has a leader page for each file. This page is used only for software checking (except for three files during the boot sequence). It is not used for recovery.

Changes to the file name table and leader pages are written to a log. The log is written such that there are two copies of all log records. Logging makes atomic update easy, delays some updates, and reduces the number of writes necessary for an update. Although logging may not be a cost-effective technique for the data of a file system, it is effective for the metadata [Need87].

*Group commit* batches a set of updates together for a log write. The log is written (if necessary) every half second. Group commit reduces the number of writes to the log.

FSD creates a one byte file by finding two free pages from the VAM, updating the file name table, and writing the leader and the data page. A file create typically does one I/O synchronously: the combination of the write of the leader and data pages. The create also dirties some file name table pages that are asynchronously logged and eventually written back to the file name table.

## 5. Design Discussion

The design described here met the file system requirements, performed the best of the designs considered (as predicted by the analysis), and scaled well to slow-seeking but high-transfer-rate disks. The principle design concept is *locality on the disk.* Information that is needed, generated, recovered, or retrieved together benefits from proximity on the disk.

### 5.1 File name table

The file name table maps a file name into a file. The file name tables in CFS and FSD are kept in a B-tree. The information about each file in CFS is split between the B-tree and the file header. In FSD, the run tables and file properties, formerly stored in the file header, are kept in the file name table.

Keeping the name and property information together is desirable for operations over many files such as "list" and "open." There is no need for a disk read for the properties since they are already available in the file name table. The file name table is preallocated to sectors near the central cylinder of the logical volume. This reduces disk head motion.

The unique identifier and the file's run table can be stored directly in the file name table since all files have (at most) one name. If files could have more than one name, such as with the multiple hard links in the UNIX file system, this optimization would be more difficult.

To improve robustness, the file name table is written twice: every page is written on two different sectors with independent failure modes. Due to the extensive buffering provided by the log (see below), the overhead for double writing is not excessive. When a page is read, both copies are read and checked.

### 5.2 Leader pages

Files in FSD consist of a single leader page and the data pages. The leader page doesn't contain any information needed for operation, but provides an optional check for the proper operation of the system. Leader pages and the file name table are different data structures that are mutually checking. Leader pages are a key element in the robustness of FSD (see section 5.8, *Robustness*, below).

### 5.3 Log-based recovery

Atomic update of metadata is a property required in file systems. However, CFS used a B-tree package that did not have atomic update. While complicated splits or joins were being done in the tree, the tree could be left in an inconsistent state by a crash. Consistency was reestablished by scavenging, although this was a slow operation (an hour or more on a 300 megabyte disk). Further, the name table pages spanned multiple disk pages and a partial write could corrupt a name table page.

A lesson learned in building database systems is that performance can be gained and consistency achieved by writing updates to a log on stable storage. Data spread over the disk can be logically and atomically updated with a single disk write to the log. Updates are applied to buffered copies of pages, but the copies are *not* forced to disk – they are just written to the log. The writes of the buffers may be delayed (once logged) in anticipation of a further update to the page (a *hot spot*) or the writes may be done at a more convenient time.

In comparison, some systems do synchronous writes for consistency (see [Bach86], sections 5.16.1 and 5.16.2, for a good discussion of the UNIX file system.) Synchronous writes require that the writes be performed in a particular order before an operation can complete (e.g., a file create in UNIX writes the inode to disk before returning). Synchronous writes tend to do more writes than logging and the writes are not localized. Logs have been used previously for file servers. Alpine [Brow85] logs updates to the file name table and file contents, but not to the allocated page map. DFS [Stur80] used a variation of logging called intention lists.

A log in FSD is used only for changes to the file name table and to the leader pages. It logs physical pages and is a redo log [Gray79]. That is, it records images of changed pages that must be (re)written if the system crashes. Logging allows atomic update of the file name table and the leader pages. Like the file name table, the log is allocated to sectors near the central cylinder to minimize head motion.

The log could also be used to record changes to the VAM and file data pages. The VAM is maintained in a volatile form (see section 5.5, *Free Pages*, below) so it does not have to be logged. VAM logging would greatly decrease worst case crash recovery time from about twenty five seconds to about two seconds. VAM logging was not done since it was a complicated modification, worse case recovery is rare, and recovery was fast enough anyway.

In CFS and FSD, data hot spots are rare. Both systems support versions for files. Most files are written exactly once. Hence, the logging of the data pages is not very attractive: logging data pages would come close to doubling the number of pages written for file data.

The log is kept as a circular disk file. New log records are appended to the log by synchronously writing the new record to the file. Since the file is circular, there are potential problems in overwriting data while it is still needed, and coordinating the writing of file name table pages and leader pages. Database systems typically use complicated algorithms to make use of almost all of the log. FSD uses a simple algorithm.

First consider just the file name table pages. FSD maintains a cache for pages of the file name table. Updates are applied to pages in the cache and then logged. The log is divided into thirds. FSD records in the cache the identity of the third where the page was last logged. When the current log write is about to enter a new third, there may be data logged in this new third that must be rewritten to disk. The only disk resident copy of the data is in the log and it will be overwritten soon. Any pages logged in this new third, but not logged in a later third, are written to the file name table by the logging code. The pages to be written are discovered by scanning the cache looking for pages that were most recently written into this new third. The cache is maintained such that the "dirty but logged" pages are kept in the cache; the write of the pages to the file name table on the disk is directly from the cache.

Due to high locality in the file name table, the number of name table pages normally written is nearly zero. It is usually the case that a dirty name table page will have been recorded in a newer third, and thus no write will be required. Leader pages are also logged. Leader pages for a file create are normally written by piggybacking the write on the next operation to the file thereby avoiding a write by the logging code. Otherwise, they are written during entry into the third where they were logged. The only times pages are written to the file name table are during entry into a new third and during crash recovery. This simple algorithm averages 5/6ths of the log in use.

Log records that are nearly the size of the log file call for drastic measures that will not be covered here, but which are handled in the system. A log entry that is longer than the log file will cause a crash, but the log is forced long before this should occur.

Any system that keeps its permanent data on disk must be concerned with disk errors. The model of disk errors used in FSD is that only one error will occur at a time and it will damage one or two consecutive sectors. With this failure model, multi-sector writes may be only partly done. When writing the last two pages, either both are transferred successfully, the last page is detectably damaged but the next to last is transferred successfully, or both pages are detectably damaged (note similarities with the *weak atomic property* in [Stur80]). Hence, it is only necessary to avoid writing replicated copies of a page into adjacent sectors. No single disk write can damage both copies.

With this in mind, the details of the log format can be described. A pointer to the start of the first valid record in the oldest third is kept in page zero (replicated in page two) of the log. The pointer is updated whenever a new third of the log is entered, after the pages in the to-be-overwritten third are written to disk. Each log entry is comprised of a header page, a blank page, a copy of the header page, the data pages being logged, an end page, copies of the data pages being logged, and a copy of the end page. The same data is never written to adjacent pages. Failure of the write at any point can be detected when the log is read by matching the start and end page copies. Single or double page errors can be corrected from the other copy. The end of the log is detected by reading a header page pair and checking log record numbers, boot count, end pages, and special bit patterns in the header page.

## 5.4 Group commit

The system also implements a variant of *group commit* [DeWi84]: a set of updates are grouped together in one log write to amortize the cost of the log write disk I/O over several updates. Where databases group the updates of independent users, FSD groups some updates of the workstation owner. FSD forces its log twice a second. This induces a certain degree of uncertainty about when some modifications to the file system are permanent, but the uncertainty is only half a second. Clients may force the log.

In the event of a crash, any workstation user must determine the state of the system anyway. Loss of up to a half a second is not significant since it is regained in increased performance of a few seconds of normal operations. The last-used-time for cached copies of remote files is an excellent example of data that does not require exact update. Although uncertainty cannot be tolerated in a database or a transaction system, it is already tolerated in workstations with existing file systems.

The use of group commit also helps with hot spots. Bulk updates are often done to the file name table [Schm82]. These updates are normally localized to a sub-directory, which fits on a few pages. These pages are rapidly dirtied by the bulk updates. By doing group commit, the log is consumed more slowly and written less often. Moreover, the name table itself is written infrequently. One benchmark measured the combination of logging and group commit as reducing the number of I/O's for metadata by a factor of 2.98 during these bulk operations; the total reduction was a factor of 2.34 for all I/O's. These factors may be improved somewhat by using a bigger log and lengthening the time between commits.

Log records vary in size depending on activity. Records have five pages of overhead and write twice the data to be logged. An open of a cached file from a file server changes the last-used-time in the file properties. If this were the only update during a group commit period, then it would be recorded as a one data page record. This is logged in seven 512 byte sectors. The longest log record observed is 83 sectors long. Under high load, a typical log record has 14 pages logged, for a log record size of 33 sectors.

## 5.5 Free pages

The free page information is kept in a bitmap called the VAM. Updates to the VAM could be made by synchronous disk writes. FSD avoids all disk writes during normal operations by keeping the VAM in volatile memory. During a controlled shutdown and idle periods, the VAM is written to disk. During a boot, the VAM is read from disk if it has been properly saved. If not, it is reconstructed from the file name table. Since the file name table is a compact structure with a great deal of locality, it can be processed quickly. The time to reconstruct the VAM on a Dorado with a 300 Megabyte file system is typically twenty seconds.

One complication in maintaining free pages is that the pages are not really free until the delete is committed. They cannot be allocated to a new file since they might then be written. Pages in deleted files are kept in a shadow bitmap. When a commit occurs, the pages marked free in the shadow bitmap are marked free in the VAM.

## 5.6 Page allocator

The File Package in Cedar allocates pages in runs (often called extents). The allocator in CFS performed adequately, except that it tended to fragment the free space. Large free blocks of space were broken up by small files. A large fraction of files are small. A measurement of one system shows 50% of files are less that 4,000 bytes but use only 8% of the sectors.

FSD partitions the disk into big and small file areas to curtail fragmentation. The areas are only hints: a big file may have pages in the little file area. This is similar to many memory allocators: dynamic storage is grown starting from small addresses, while the stack is grown from the end of memory towards small addresses.

This allocator should work very well in FSD. Most of the small files are cached copies of files stored on file servers. The size of these files are known when they are fetched and the sizes never change. New versions of files may be cached, but old versions are immutable (except that they may be flushed).

## 5.7 File open

Opening a file does not usually require an I/O. The leader page is normally verified on the first access to a file by piggybacking its read with the access. The first data access is almost always to the first data page, and the leader page is the previous physical page on the disk. Hence, it usually costs only the transfer time for a page to read the leader page.

## 5.8 Robustness

Using different data structures and algorithms is a well known method to detect bugs: both CFS and FSD use this technique. Leader pages have detected many bugs in FSD. However, leader pages are not as effective as the headers and labels of CFS. Labels checked nearly every file system I/O. In FSD, bugs in the page allocator, logging, or crash recovery cannot be detected when they occur. The bugs are detected later, but they are harder to track down and may have done damage to the file system. FSD keeps pages cached from the file name table as read-only except when they are being updated. This is to catch wild stores, but this has never occurred.

CFS rarely took label errors that were due to incorrect software. The scavenge program did not read the labels to verify the run tables stored in the headers. Hence, the header and label redundancy was not fully exploited. The amount of code that *must* be correct to maintain minimal system consistency has been increased from about four pages to ten pages. From analyzing system failures and measuring the system, it was estimated that elimination of the header and label redundancy would have few adverse effects.

FSD when compared to CFS is robust against six additional types of errors. First, multi-page B-tree updates were not atomic. Second, a partial write of the file name table could produce an inconsistent page. Logging prevents both of these. Note also that the log writes two copies of all pages. Third, the file name table could have bad pages; it now is replicated. Fourth, the VAM can have disk errors; these are recovered by reconstructing the VAM. Finally, two kinds of pages needed in booting could become bad; they are now replicated.

### 5.9 Recovery

Recovery is fast and easy. There are two types of recovery. First, the VAM can be reconstructed using the name table (see section 5.5, *Free Pages*, above).

Second, the file name table and labels are recovered from the log. The log is a physical redo log and the algorithm to perform recovery is simple. Log records are read and the copies of pages in the log are written to disk. Recovery rarely takes more than two seconds on the current hardware.

Scavenge in CFS was infrequent but very time consuming. Users do not like their machines being unavailable for an hour or more. Although in principle the replication and recovery in CFS protects the file system, the lack of locality of the data structure makes recovery too expensive.

## 6. Performance Analysis

Why choose one design instead of another? How much performance does a feature deliver? How much does replication cost? How do existing systems perform? One way to answer questions like these is to construct a model.

The model used in the design of FSD computes the expected average case times for typical file operations. These operations included create, delete, list, open (without data I/O), and recovery from a disk error. All models used caches for all disk resident data. The caches were assumed to hit if the information is small (e.g., in the VAM), and to hit except for the leaf nodes for large structures such as the file name table. Hits for leaf nodes were modeled by simple probability distributions.

In the design of file systems, it is common to use the estimated number of I/O operations as the performance metric. Unfortunately, this metric does not capture the rotational and radial position of the disk heads. In particular, lost revolutions, sector clustering by cylinder, read then immediate re-write of sectors, and short seeks are not adequately modeled.

Each design alternative for FSD was analyzed in terms of its effect upon each operation. The numbers of seeks, short seeks (a few cylinders), latencies (half a revolution), lost revolutions, and transfer time were estimated by analyzing and *scripting* the necessary operations. The scripts incorporated any known locality, both rotational and radial (e g., dropped revolutions and same cylinder seeks). It was assumed that there would be no interference in using the disk. Estimates were made for both hitting in the file name table cache and for missing.

The idea is quite simple. Based on the code or documentation, analyze the algorithm to find out where it will do I/O's. If an I/O will be on the same (or nearby) cylinder or if the rotational position of the disk is known, then take this rotational and radial position into account in computing the time for the I/O. Compute both the cache hit and cache miss cases, and compute a weighted average.

By way of explaining the script method, here is an example of the first three entries in a script that creates a one sector file in CFS:

1) Verify free pages: 1 seek, 1 latency, 3 page transfers

2) Write header labels: (revolution - 3 page transfers), 2 page transfers

3) Write data labels: revolution, 1 page transfer

4) ...

The file needs three pages: two for the header and one for data. Free pages are found in the VAM without incurring an I/O. The pages have to be verified as free, so a seek, latency, and a three page transfer to read labels is performed (1). Assuming the pages are really free, then the labels on the header are written (2). The time for the write starts from the end of (1) and is the time of a disk revolution less the time for a three page transfer, and it takes two transfer times. The two sectors are the first two verified in (1) and they have just gone past the disk head. Finally, write the label for the data sector (3).

This model was validated by estimating and measuring performance of CFS, 4.3 BSD UNIX, and two types of file servers. For the simple operations benchmarked, the model almost always predicted performance to within five percent of measured performance.

Many alternatives were examined using the model. The poorer alternatives were quickly discarded. The model allowed estimation of the effects of logging, group commit, redundancy, and central placement of certain files.

A problem with this model is that it ignores CPU time. As a result, the design selected was very stingy with disk I/O's, but the CPU was sometimes a slight bottleneck. The Dorado is a high performance workstation with somewhat slow, older technology disks. Cedar is a system that uses lightweight processes, single virtual memory, low overhead monitors, and is quite efficient in the use of the CPU. Faster CPU's such as the Dragon will be common in workstations as will slower disks (e.g., optical disks). The combination of these factors led the author to ignore the CPU in the modeling, although this may not be a proper assumption for all environments.

## 7. Performance

CFS and FSD were benchmarked. Table 2 shows the timing and speed up of some common operations. All creates, opens, and deletes are for different files in the same directory. Note that the "read page" time is identical in both systems: the disk hardware is the same, so a simple file read takes the same amount of time, once the file is open. Typically, programs that are file system intensive have improvements from 25 to 50% in running time, but some operations have improved by a factor of 5 or even 100. Table 3 compares the disk I/O's of CFS and FSD. The MakeDo program used as a benchmark is typical of clients that intensively use the file system. "Crash recovery" is the time it takes to recover from a major crash on a moderately full 300 megabyte file system.

| | CFS | FSD | Speed up |
|---|---|---|---|
| Small create | 264 | 70 | 3.77 |
| Large create | 7674 | 2730 | 2.81 |
| Open | 51.2 | 11.7 | 4.38 |
| Open + Read | 68.5 | 35.4 | 1.94 |
| Small delete | 214 | 15 | 14.5 |
| Large delete | 2692 | 118 | 22.8 |
| Read page | 41 | 41 | 1.0 |
| Crash recovery | 3600+ sec | 25 sec | 100+ |

**Table 2. CFS to FSD Performance Measured in Wall Clock (times in msec)**

| | CFS | FSD | Ratio |
|---|---|---|---|
| 100 small creates | 874 | 149 | 5.87 |
| list 100 files | 146 | 3 | 48.7 |
| read 100 small files | 262 | 101 | 2.69 |
| MakeDo | 1975 | 1299 | 1.52 |

**Table 3. CFS to FSD Performance Measured in Disk I/O's**

Table 4 attempts to compare FSD and a 4.3 BSD UNIX system running on a VAX-11/785. The "time" command measured the number of disk I/O's for the 4.3 BSD UNIX file system. Note that 4.3 BSD does not double write the directories or the inodes, so it is doing less work for a create than FSD. Table 5 compares the CPU and disk bandwidths that can be delivered by 4.2 BSD (taken from [McKu84]) and the same values for FSD. One further point of comparison is crash recovery. PARC's VAX-11/785 recovers in about seven minutes (using fsck) while FSD takes 1 to 25 seconds. Both systems have 300 megabyte file systems that are moderately full.

| | FSD | 4.3 BSD | Ratio |
|---|---|---|---|
| 100 small creates | 149 | 308 | 2.07 |
| list 100 files | 3 | 9 | 3 |
| read 100 small files | 101 | 106 | 1.05 |

**Table 4. FSD and 4.3 BSD Performance Measured in Disk I/O's**

| | FSD | | 4.2 BSD | |
|---|---|---|---|---|
| | % CPU | % Bandwidth | % CPU | % Bandwidth |
| read | 27 | 79 | 54 | 47 |
| write | 28 | 80 | 95 | 47 |

**Table 5. FSD and 4.2 BSD Performance Measured in Percent of CPU and Disk Bandwidth**

The implementors of CFS knew how to build a faster system, but their goals were to investigate other ideas. Hence, the comparisons in Tables 2 and 3 are somewhat exaggerated. Table 4 shows that creates in FSD use about half of the I/O's used by 4.3 BSD. Inodes in 4.3 BSD are located on the same cylinder group as their directory (when possible). A disk read fetches several inodes. The benchmark favors 4.3 BSD since all the files were in the same directory. Hence, the disk traffic for inodes is fairly small for listing and reading 100 files.

## 8. Conclusion

FSD meets its design goals. It is robust, yet it does not use labels. It is high performance, rarely doing unneeded disk I/O's. In operations on the structure of the file system (open, delete, extend, contract, and list), it rarely does any disk I/O's; it is mostly CPU bound. It has fast recovery.

In addition, it has four atypical aspects. First, it was designed using a performance model that captures most of the timing characteristics of disks. Second, it uses log-based recovery. Third, group commit is used to decrease disk traffic. The combination of these last two allows for delayed write of many pages, so the I/O to many hot spots can be reduced. Finally, the system performs double writes of key system structures. The performance penalty for these writes is not large, due to the decrease in traffic from logging and group commit.

Workstation file systems can be built using some techniques from database systems. Although these techniques may not be cost-effective for the data of a file system, they are effective for the metadata. These systems can be robust, have high performance, and recover rapidly from a crash.

## Acknowledgments

The comments from the program committee and referees helped the author to improve the content and presentation of this paper. Particular thanks go to Michael Schroeder. Carl Hauser provided valuable assistance in the design of the log file format. Roger Needham, Carl Hauser, Jules Bloomenthal, and Douglas Terry provided valuable comments on this paper. Subhana Menis provided her usual excellent editorial assistance.

The principal original authors of the File and FS packages were Andrew Birrell and Michael Schroeder. It was comparatively easy to make the modifications outlined in this paper due to way they wrote these packages.

## References

[Bach86] Bach, M. J. *The Design of the UNIX$^{TM}$ Operating System.* Prentice-Hall, Englewood Cliffs, 1986.

[Brow85] Brown, M., Kolling, K., and Taft, E. "The Alpine File System," in *ACM Transactions on Computer Systems.* Vol. 3, No. 4 (November 1985), 261-293.

[DeWi84] DeWitt, D. J., Katz, R. H., Olken, F., Shapiro, L. D., Stonebraker, M. R., and Wood, D. "Implementation Techniques for Main Memory

Database Systems," *Proceedings of SIGMOD '84*, June 1984, 1-8; also appears in *SIGMOD Record* Vol. 14, No. 2.

[Gray79] Gray, J. "Notes on Data Base Operating Systems," in *Operating Systems, An Advanced Course*. Springer-Verlag 1979.

[Lamp79a] Lampson, B. W., and Sproull, R. F. "An Open System for a Single-User Machine," *Proceedings of the Seventh Symposium on Operating Systems Principles*, Dec. 1979, 98-105.

[Lamp79b] Lampson, B. W., and Sturgis, H. E. "Crash Recovery in a Distributed Data Storage System," Xerox PARC CSL unpublished working paper, Palo Alto, CA, 1979.

[Lamp81] Lampson, B., and Pier, K.; Lampson, B., McDaniel, G., and Ornstein, S.; Clark, D., Lampson, B., and Pier, K. *The Dorado: A High Performance Personal Computer. Three Papers.* Xerox PARC Report CSL-81-1, 1981.

[McCr84] McCreight, E. "The Dragon Computer System: An Early Overview," in *Proceedings of the NATO Advanced Study Institute on Microarchitecture of VLSI Computers*. Urbino, Italy, July 1984.

[McKu84] McKusick, M. K., Joy, W. N., Leffler, S. J., and Fabry, R. S. A Fast File System for UNIX," in *ACM Transactions on Computer Systems*. Vol. 2, No. 3 (August 1984), 181-197.

[Metc76] Metcalfe, R. M., and Boggs, D. R. "Ethernet: Distributed Packet Switching for Local Computer Networks," *Communications of the ACM*, Vol. 19, No. 7 (July 1976), 395-404.

[Need87] Needham, R. M. Private communication.

[Rede80] Redell, D., Dalal, Y., Horsley, T., Lauer, H., Lynch, W., McJones, P., Murray, H., and Purcell, S. "Pilot: An Operating System for a Personal Computer." *CACM*, Vol. 23, No. 2 (Feb. 1980), 81-92.

[Ritc78] Ritchie, D. M., and Thompson, K. "The UNIX Time-sharing System." *Bell System Technical Journal*, Vol. 57, No. 6 (July-Aug. 1978), 1905-1930.

[Schm82] Schmidt, E. *Controlling Large Software Development in a Distributed Environment*, Ph.D. Thesis, U.C. Berkeley 1982; also available as Xerox PARC Report CSL-82-7, 1982.

[Schr85] Schroeder, M. D., Gifford, D. K., and Needham, R. M. "A Caching File System for a Programmer's Workstation," *Proceedings of the Tenth Symposium on Operating Systems Principles*, Dec. 1985, 25-34.

[Stur80] Sturgis, H., Mitchell, J., and Israel, J. "Issues in the Design and Use of a Distributed File System," *Operating Systems Review*, Vol. 14, No. 3 (July 1980), 55-69.

[Swin86] Swinehart, D., Zellweger, P., Beach, R., and Hagmann, R. "A Structural View of the Cedar Programming Environment," in *ACM Transactions on Programming Languages and Systems*. Vol. 8, No. 2 (October 1986), 419-490; also available as Xerox PARC Report CSL-86-1, 1986.

[Thac79] Thacker, C., McCreight, E., Lampson, B., Sproull, R., and Boggs, D. *Alto: A Personal Computer.* Xerox PARC Report CSL-79-11, 1979.