HIGH PERFORMANCE FILE SYSTEM DESIGN

Carl Hudson Staelin
(Thesis)

CS-TR-347-91

October 1991

HIGH PERFORMANCE FILE SYSTEM DESIGN

Carl Hudson Staelin

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE

October 1991

Dedication

In memory of Virginia Erhardt Mahoney

Acknowledgements

I would like to thank all those who made this thesis possible and made my stay in Princeton enjoyable. The Computer Science Department at Princeton has provided a stimulating environment. In particular, I would express my appreciation to Rafael Alonso, Matt Blaze, Chris Clifton, Luis Cova, Mordecai Golin, and Christos Polyzois, for their advice and criticism given over the course of this work.

In addition, I would like to thank Amdahl Corporation for providing the impetus that started my thesis research. In particular I would like to express my gratitude to Dieter Gawlick and Dick Wilmot for suggesting that I visit Amdahl. I would also like to thank Dieter Gawlick for his continuing interest in my work and for his helpful advice.

I would like to thank my advisor, Hector Garcia-Molina, for his advice, patience, and friendship, and my readers, Kai Li and Dave Hanson, for their many useful suggestions and comments. Finally, I would like to thank Sigal Ar and my family for their support and encouragement.

Abstract

File systems and I/O subsystems should be *smart*; they can analyze how they are being used and tune themselves dynamically to improve their performance. File systems should select caching and disk placement strategies on a per-file basis, and they should use system-wide disk reorganization strategies. For example, systems should be able to reorganize the data on disk automatically during idle periods so that system performance is improved during future periods of peak load.

This dissertation presents the design and analysis of iPcress, a prototype of a next-generation file system. iPcress is a smart, high-performance, reliable file system. It uses statistical information collected on a per-file basis to tune itself. iPcress has a framework in which various optimizations can be performed by the file system automatically. It is extensible; other optimization techniques can be incorporated easily, so that the system may evolve. In addition, iPcress can incorporate a variety of file access and placement techniques and choose the best combination of techniques for each file dynamically.

A sample smart optimization — clustering active disk data in the center of the disk — is described; it increases disk throughput up to 30%.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

File systems and I/O subsystems should be *smart*; they can analyze how they are being used and tune themselves dynamically to improve their performance. File systems should select caching and disk placement strategies on a per-file basis, and they should use system-wide disk reorganization strategies. For example, systems should be able to reorganize the data on disk automatically during idle periods so that system performance is improved during future periods of peak load.

This dissertation presents the design and analysis of iPcress, a prototype of a next-generation file system. iPcress is a smart, high-performance, reliable file system. It uses statistical information collected on a per-file basis to tune itself. iPcress has a framework in which various optimizations can be performed by the file system automatically. It is extensible; other optimization techniques can be incorporated easily, so that the system may evolve. In addition, iPcress can incorporate a variety of file access and placement techniques and choose the best combination of techniques for each file dynamically.

1.1 Motivation

Trends in hardware development affect the basic price/performance tradeoffs that are made in operating system design. Operating systems may have to be redesigned to take advantage of rapid improvements in hardware. The three most important trends are the dramatic improvements made in CPU performance, the lack of any significant improvements in secondary storage (disk) performance, and the steady drop of memory prices.

Consequently, general-purpose machines are becoming increasingly I/O limited. Most

I/O is done through a file system, so we focus on improving file system performance. Also, most common file systems were designed 10–20 years ago, with different concerns and performance tradeoffs than today, so there is potential for improvement.

For example, file systems generally use one technique for caching in memory and storing data on disk. However, there are many strategies, each of which is best in different circumstances, and next-generation file systems should use different techniques for different circumstances. For example, sequential scans of large files are handled best by contiguous allocation of disk space and prefetching data from disk into the cache. Conversely, random accesses to a large file are handled best by fetching data only on demand.

1.2 Outline

Chapter 2 describes issues in file-system design, prior file-system designs, and other results related to I/O subsystems. It surveys the current hardware environment, presents the tradeoffs involved in file system design, and describes successful and unsuccessful features of existing file systems.

Chapter 3 analyzes MVS[1] file-access patterns based on file-system trace data. This analysis and trace data show that some file-access statistics can predict future file-access patterns. In particular, file-access patterns are skewed; there are a few active files and many inactive files, and the active files stay active while the inactive files stay inactive.

Chapter 4 details the design of the iPcress file system, which is based on both the results from Chapter 3 and on the prior experience and analysis presented in Chapter 2. In iPcress, *everything* is a file, and placing file system meta-data in files simplifies design and may improve both performance and reliability. We also describe how caching and layout algorithms may be selected independently on a per-file basis. Chapter 5 presents the implementation of iPcress, focussing on its internal structure, limitations, and ways to reduce or eliminate these limitations.

Chapter 6 compares iPcress's performance to the performance of the ULTRIX 3.3 file system using several standard benchmarks. iPcress provides performance similar to that of ULTRIX, even though iPcress is a user-level process. In particular, iPcress outperforms ULTRIX, except for sequential scans of large files.

Chapter 7 presents a "smart" optimization — clustering the active data in the center

[1] MVS is an IBM operating system for mainframe computers.

of the disk, and describes how this optimization was added to iPcress. We model and simulate disk clustering to demonstrate expected performance benefits and to compare the effectiveness of various clustering algorithms. Adding the clustering algorithm to iPcress improved disk performance 10–30%.

Some of the material presented in this dissertation has been published previously. Material from Chapter 3 appeared in Reference [110]. Chapters 4 and 5 contain material from References [112] and [113], and Chapter 7 contains material from Reference [111].

Chapter 2

Prior Work

In order to design a high-performance file system, we must first understand how existing file systems are used, so we analyze file-access patterns in existing systems. We describe several file systems and categorize their design decisions, and we describe the common methods for evaluating file-system performance.

2.1 File Access Patterns

There has been much research on file system use. We are interested in specific data that may help predict file-access patterns and, in general, statistics such as read/write ratios. This information is useful because there are optimization strategies that can be used once future activity can be predicted reliably. In particular, strategies that are most effective only under specific conditions could be used, if the data revealed that those conditions actually occur.

Briefly, files tend to be accessed either heavily or not at all, and files tend to be accessed sequentially. Also, files tend to be only read or only written.

2.1.1 Skewed File-Access Patterns

File access patterns are highly skewed, i.e., certain files receive far more I/O than others [8, 21, 42, 119]. This result is important because it implies that improving performance for the small fraction of active files should improve system performance. Wilmot [119] showed that many file systems obey a 90/10 rule, and more interestingly, they also obey a 50/1 or 25/1 rule, where an x/y rule means that $x\%$ of the activity is absorbed by $y\%$ of the

file-system space. Improving performance for 1% of the file system optimizes 25–50% of the I/O activity. Furthermore, Wilmot's results include only files that were accessed during a specific trace period, which is a small subset of the total file system. Baclawski [8] showed that file skewness seems to be self-similar: if the file system obeys a 60/30 rule, then the hot 30% also obeys a 60/30 rule. Baclawski found that the whole file system obeys an 88/33 rule, but self-similar subsets obeyed 60/30 rules.

Most files in UNIX file systems are inactive; only 3.6–13% of the file-system data is used in a given day, and only 0.2–3.6% of the I/O activity goes to the least active 75% of the file system [21]. Others found that 66% of user files are not accessed for a month [42], which implies that all user activity was focused on at most 34% of the file system.

Further evidence that most files are dormant is provided by Alsoft [30]. Their disk reorganization utility for the Apple MacIntosh, DiskExpress II, reorganizes the disk based on file activity statistics collected during normal system use. Alsoft found that roughly 80% of disk space is inactive even in a full file system.

2.1.2 Sequential Access

Most file accesses are sequential and transfer an entire file. Three studies demonstrate that file access in UNIX tends to be sequential. Reference [42] shows that the type of the file (e.g., temporary, permanent, or system log) also influences its access pattern.

In Reference [85], 66–68% of the data transfers observed in the traces were sequential, 91–93% of read-only opens accessed data sequentially, and 96–98% of write-only opens accessed data sequentially. These figures include opens, which execute a single seek before accessing any data in the file. Only 19–35% of read-write accesses were sequential, but the majority of opens are read-only or write-only.

Reference [69] describes the analysis of several UNIX machines with various workloads. Most I/O was sequential, but certain applications (e.g., the loader) always access certain files randomly.

Reference [42] reports the analysis of sequentiality by file type, such as temporary, permanent, or accounting log. Sequentiality was strongly dependent on the type of the file, how it was opened (e.g., read-only), and the size of the file. However, 68% of all files and 94% of user files opened for read-only access are read completely.

2.1.3 Open Mode

Files tend to be either read or written [17]; files that have been read recently tend to be re-read in the future, and files that have been written recently tend to be overwritten in the future.

These results suggest that file systems may use different algorithms to store and access read-only and write-only files. Optimization strategies for each case may provide dramatic performance improvements over existing approaches. Strategies for each case are usually very different, so file systems may improve their performance by categorizing each file and using the appropriate strategy.

2.2 Overview of Existing File Systems

Nearly all current UNIX file systems have shortcomings. One problem is that most file systems are limited to a single device; files cannot span multiple devices and the system cannot use multiple devices for redundancy to provide reliable service.

By limiting themselves to a single device, most file systems increase system administration tasks and limit reliability and performance. There are well known techniques, such as mirroring and checksumming [16, 51, 52], that improve file system reliability dramatically but require multiple disks. Other techniques, such as load balancing and data striping [96], optimize throughput for a set of disks. Some attempts have been made to add mirroring to UNIX invisibly, but, for the most part, file system reliability in UNIX is accomplished through the assiduous collection of backups.

Distributed file systems face many of the same problems as traditional (non-distributed) file systems. In particular, caching strategies are particularly important in distributed systems.

2.2.1 Non-Distributed File Systems

The UNIX file system [81] is known for its poor performance and questionable reliability [94]. The Berkeley Fast File System, which has better performance than the original file system, uses a simple, variable block-size scheme with two block sizes, the larger of which is often 4–8 Kbytes. This scheme provides poor performance because large files still require separate disk accesses for each block. UNIX uses a demand-driven block cache, with a simple scan

detection scheme that tries to improve read performance for sequential scans by reading the next block in the file if the last block accessed is the current block's predecessor.

In UNIX, files are streams of uninterpreted bytes. A file is named by its system identifier, which is an index into a table of file header records known as *inodes*. The inode contains information about the file, such as its size, ownership, access permissions, and location on disk. Each file may have several names; directories associate names with inodes. Each file system has its own inode table, and file identifiers are unique only within a file system.

The UNIX file system has no intrinsic reliability or recovery mechanisms. If there is a device failure, the only means of recovery is restoring the file system from a dump. In addition, when a UNIX machine crashes, the file system may be left in an inconsistent state. Consequently, the entire file system must be checked and cleaned up after each crash. This process is time consuming and makes the time required to reboot a linear function of the disk space.

The Dartmouth Time Sharing System (DTSS) used an extent scheme based on the buddy system [78]. Most files occupied a single extent, and, when necessary, files were split into several extents in order to reclaim space lost to internal fragmentation. On the average, each file had 1.5 extents. Over 70% of the files were contained in 2 Kbyte sectors, which is the smallest extent. Roughly 98% of I/O requests were satisfied by a single I/O operation, but the run-time library used 2 Kbyte buffers so that all run-time library requests would be satisfied by a single I/O operation.

The RAID file system is not really a file system but rather a technique for making an array of small disks look like a single large disk [50]. The advantages of this approach are that a large, sequential access has a throughput equivalent to the sum of the throughputs of all the disks and that disk reliability can be improved by using uniform encoding techniques to recover from device failures. Unfortunately, this approach does not work well with small random accesses, which occur when accessing many small files.

A completely different approach is used by the log-structured file system [84, 94]. This file system consists entirely of a database-like log. In conjunction with a large cache, this approach offers high write performance because devices rarely seek and writes operate near device limits by writing large blocks. This approach improves write performance by (potentially) sacrificing read performance. They argue that read performance is not critical, because large disk caches reduce disk read traffic dramatically.

An intermediate approach is used in IBM's AIX version 3 Journaling File System (JFS)

and Logical Volume Manager (LVM) [22, 82]. LVM provides RAID-like capabilities in software and allows a collection of disks to masquerade as a single (perhaps larger) disk. It allows the logical volume to include mirror disks. JFS is a UNIX file system that uses a short-term database log on disk for quick recovery from failures.

2.2.2 Distributed File Systems

Distributed file systems must cope with delays due to network bandwidth and latency, so clients must cache file data. In addition, many distributed file systems attempt to reduce file server load by minimizing client-server interactions. All of the file systems use some form of data cache on the client. Most file systems attempt to mimic the UNIX file system semantics regarding shared data coherence, but some (such as the Andrew file system) do not.

Sun Microsystem's Network File System (NFS) was one of the first network file systems to gain wide acceptance. It uses a block-oriented cache and ad-hoc consistency control. The primary advantages of NFS are that it is widely used, is simple to implement and understand, and has acceptable performance. NFS uses a stateless interface, which simplifies distributed failure recovery at the cost of higher client-server interaction. Unfortunately, NFS requires a great deal of client-server communication, which limits the number of clients for each network file server.

The Amoeba file system, called the Bullet File Server [116], uses file caching rather than more traditional block caching. All files are contained in contiguous space both in the cache and on disk, and data transfers from server to client involve whole files. The Bullet file system performance figures are impressive: roughly 3 to 8 times better performance than NFS both in terms of bandwidth utilization and reduced delay.

The Andrew file system [67] minimizes client-server communication whenever possible. It caches files on the client and uses a token-based consistency strategy to reduce client dependency on the server. The client does not need to communicate with the server when it holds the file and the file's token. Tokens confer read-only or read-write permission. Modified files are copied back to the server only when the file is closed, which also reduces client-server interaction. However, the server must keep track of all tokens because it may have to revoke them.

Another network file system, Sprite, uses file caching, and compared to NFS, Sprite runs various benchmarks faster and decreases server load. Sprite also uses a workstation's

local disk to cache files to minimize network and server load. Sprite minimizes consistency protocol overhead using a lease-based token system [54]. Leased tokens expire after a given time interval, which can reduce the number of token revocations issued by the server.

2.3 File System Design

2.3.1 Hardware

The most common form of secondary storage is the disk drive, although many devices, such as removable disks, reel-to-reel tapes, cartridge tapes, write-once-read-many optical jukeboxes, electronic disks, and removable read-only optical drives, are commonly available. These devices may generally be classified as rotating media, streaming media, or electronic media.

Rotating media include disk drives, removable disk drives, optical jukeboxes and removable optical drives. There is a rotating platter, like an LP record, that stores the data on either magnetic or optical medium. A read/write head, like the record player's stylus, is positioned radially over the platter. Physical delays for completing an I/O include moving the read/write head to the correct radial position, and waiting for the platter to rotate so that the data is under the head. The delay incurred by moving the head to the correct radial position is the *seek* delay, and the delay incurred waiting for the platter to rotate to the correct position is the *rotational* delay. Finally, the *transfer* delay is the time it takes to read/write the data from/to the device. Jukeboxes also incur a *medium* delay whenever the jukebox selects a new platter. Modern disks typically have multiple platters mounted on the same spindle, with a read/write head for each surface, and the heads are physically connected so that they move in unison. Most disks limit the number of heads that may be active concurrently to one or two, but there are some that allow all heads to be active. Table 1 lists some representative delays for various media.

Streaming media include various forms of tape devices. In this case, the physical delay is just the seek delay and the transfer delay. However, the seek delay is generally far greater than the seek delay for rotating media, e.g., seconds instead of milliseconds.

Electronic media are essentially a non-volatile memory bank with a disk interface. Their primary advantage is minimal (essentially zero) delay on all accesses. However, they are exorbitantly expensive to use as the basic storage medium, and they are used most often for very active (or performance critical) data.

Device	Delay (ms)			
	Medium	Seek	Rotational	Transfer (512 bytes)
Electronic Disk	N.A.	N.A.	N.A.	0.10
Fast Disk	N.A.	8	5.5	0.10
Average Disk	N.A.	16	8.3	0.25
Optical Jukebox	10,000	60	8.3	0.50
Digital Audio Tape	N.A.	30,000	N.A.	2.00

Table 1: Sample Device Delays

The overall delay experienced by a user is the sum of the physical delays combined with various software overhead and communication and protocol delays. While these latter delays are not negligible, for the most part, they can be treated as a single, fixed overhead for each access.

2.3.2 Disk Allocation

There are two basic approaches to secondary storage allocation: blocks and extents. For extents, each file's data is contained in a single, contiguous region of secondary storage called an extent. For blocks, each file's data is contained in a set of fixed-size regions called blocks. There are, of course, variations on these schemes, which are detailed below.

Existing file systems cover a spectrum of possible implementations, from pure extent systems to pure block systems. The essential tradeoff involved in choosing between fixed block size and extents is wasted disk space versus performance. Using a small block size ensures that the total space wasted to internal fragmentation (unused space within allocated blocks) is minimized, while extents and large block sizes ensure high performance for sequential I/O. Large block sizes waste disk space due to internal fragmentation, while extents waste space due to external fragmentation (no single free block large enough).

The original UNIX file system [91] uses fixed-size blocks. Unfortunately, this file system was designed when CPU speeds more closely matched disk speeds, and so they chose a tiny blocksize (512 bytes). As CPU speeds increased, this design decision limited system performance because the file system was too slow; it could use only about 3% of the disk I/O bandwidth available [81].

At the other end of the spectrum is the IBM MVS file system, which is extent based. When a file is created, the user specifies the expected size of the file, and (if possible) the

system allocates a single extent for the file. The MVS file system does allow files to grow beyond the original allocation by allocating additional extents for the file, which allows the system to create large files when there is not enough contiguous space large enough to hold the file. However, files may have at most 16 extents, which limits both the creation of large files on a fragmented disk and the "unlimited" growth of files beyond their original allocation.

An intermediate solution is to use variable size blocks. In particular, this approach has been used in the Dartmouth Time Sharing System's (DTSS) file system [78] and the Berkeley Fast File System for UNIX [81]. The DTSS file system uses a variant of the buddy system allocation scheme [76]. Disk space is allocated in block sizes that are powers of 2, and files may span several blocks.

The Berkeley Fast File System [81] uses a complicated blocking scheme. The largest block size is fixed for each file system and is usually either 4 or 8 Kbytes. In addition, each block may contain up to 8 *fragments*, which are actually blocks within a block. There are complicated rules on how these fragments may be created and allocated, but essentially these fragments may be used for small files as if they were fully independent blocks with variable size (up to the regular block size). In addition to having a limited variable size block scheme, the Berkeley Fast File System attempts to cluster blocks from a given file close together. It does so by dividing the disk into *cylinder groups*, which each have their own free list, and by allocating all blocks for a file from a given cylinder group whenever possible. It also attempts to place all the blocks in a file at rotationally optimal positions so that sequential access proceeds with minimal rotational delay. The optimal rotational spacing between blocks depends on the relative speeds of the processor and the disk.

2.3.3 Caching

Caching strategies have two parts: staging (or fetching) algorithms and replacement (or flushing) algorithms. Staging algorithms copy data into the cache from lower levels in the storage hierarchy, and replacement algorithms flush stale data from the cache to make room for newly promoted data.

Caching may be done at several layers of the system: the processor, the controller or channel, and the disk [108]. In many systems, caching is done at more than one layer. For example, UNIX systems contain the *buffer pool*, which is a global disk cache [4]. In addition, many modern disks have *track buffers*, which cache whole tracks of data. However,

we confine ourselves to algorithms for the global disk cache.

The design of efficient caching strategies rests on careful analysis of the reference patterns. File-access patterns are described in Section 2.1, and analyzed in Chapter 3. Briefly, in UNIX, most files are small, files tend to be accessed sequentially, and most files are accessed rarely.

Small files generally fit in a few disk blocks while large files generally fit into several disk blocks. In an extent-based file system, large files may be defined as those files where the time required to transfer a file is dominated by the physical transfer time rather than by seek and rotational delays.

For small files, it is best to stage and flush the whole file. In addition, if the system can predict that the file is likely to be accessed in the near future (e.g., if it has just been opened), then it is probably worth prefetching the file.

Cache strategies for large files are more complicated. Large file use may be divided into three general categories: sequential scan, append, and random access. For large sequential accesses, the best performance and cache utilization occurs when the system prefetches data in front of the scan, and flushes data from the cache behind the scan, which is done in some disk controller caches for large mainframes [115]. For append accesses (typical of log files), it is best to cache only the last block of the file. Random access of large files is probably best handled by demand staging, which stages data into the cache when it is needed, and LRU replacement, which flushes the least recently used block from the cache.

The most common form of staging algorithm is demand-based staging, where data is brought into the cache only when it is used. The advantage of demand staging is that only data that is accessed is contained in the cache, but the user must wait for each piece of data not yet accessed to be read from the disk.

In addition to demand-based staging, some systems use prefetching, which brings data into the cache when the system expects that it will be needed in the near future. The primary advantage of prefetching is that the system reads data from the disk *before* the user requests the data, with the risk that it will read data that is never used. Consequently, the system should try to prefetch data when there is a high probability that the data will be used. There are sophisticated prefetching algorithms in database systems that adaptively modify a "prefetch window" based on the sequentiality of access to the data [90, 106]. In addition, disk track buffers are a simple, transparent form of prefetching.

Since caches have limited sizes, data must be flushed from the cache in order to make

room for new data, and replacement algorithms are responsible for deciding which data to flush. The most common form of replacement strategy is the least-recently-used (LRU) algorithm. There are alternatives to the LRU algorithm, such as frequency-based replacement [93]. In this scheme, a count of the number of times a buffer is used is kept for each buffer, and the buffer with the lowest count is flushed from the cache. There are also hybrid algorithms, which combine elements of both algorithms to provide better performance than either algorithm alone [93].

One implicit assumption in the description of the LRU algorithm is that elements have the same size. Some systems, such as IBM's Transaction Processing Facility (TPF), use variable size cache blocks [68]. There are a variety of methods used to manage caches with variable size elements. The simplest approach, which is used in TPF, is to divide the cache into several smaller buffer pools, each with a fixed element size. The disadvantage of this approach is that the buffer pools are independent, so one buffer pool could be thrashing while other pools are idle. One solution to this problem is to manage buffer pool size dynamically based on each pool's recent performance.

A second approach is to maintain a single buffer pool, but to manage elements whose size is the greatest common divisor of all buffer sizes (usually the smallest buffer size) [105]. When a larger buffer is requested, the cache flushes the oldest block of contiguous buffers that have the correct size. It does so by walking up the LRU stack marking buffers as free until it finds a contiguous set of free buffers.

As an example of a real system, the UNIX file system uses a combination of demand and prefetch staging and LRU flushing. It uses block-oriented caching and a single global buffer pool, demand staging when necessary, and a simple, one-block look-ahead algorithm for multiple-block files [4].

2.3.4 Reliability and Recovery

Reliability implies that the file system can recover from interrupted operations and that it can restore (stable) data lost through device failure. In order to recover complete operations and erase incomplete operations after system failure, the database community defines *transactions*, which are atomic sets of operations [14, 62]. In order to restore data through device failure, the system replicates data.

There are two standard techniques for implementing transactions: logging and shadow pages [14, 28, 29, 62, 104]. In file systems, a transaction might encompass a single operation

(e.g., create, write, or delete). Log-based transaction systems record all actions in a stable log. The log records can contain both information used to undo incomplete operations and information used to redo completed operations. The log also records when operations complete, so the system can decide which operations to undo and which to redo during recovery. Transaction systems based on shadow paging create a separate copy of all modified pages, and, when a transaction completes, the system atomically updates the index to reflect the new location of the data. If an operation is interrupted before it completes, the index points to the old data, and the data from the incomplete operation is ignored.

Both logging and shadow paging have been used in a variety of systems. For example, the Episode file system [73] and the Cedar file system [63] both use logging to recover system meta-data. Also, the IBM AIX 3.0 file system uses logging to recover completed operations [22]. The IBM VM file system uses shadow pages to update its file system.

The advantage of logging over shadow paging is that the logging device may provide minimal latency for writes because the disk head rarely needs to be repositioned. The disadvantage of logging is that each write is done twice, once to the log and once to the file. The primary disadvantage of shadow paging is that, over time, files tend to spread out over the device because of the randomizing actions of the shadowing mechanism.

A standard technique used to recover data lost through media failure is a *mirror disk*. The principle of mirrored disks (or the more general case of *shadow sets*) is to have duplicate copies of the data on two (or more) disks [16]. Recovering data after media failure is simple — just read the data from the mirror disk. However, there is significant cost in "wasted" disk space.

Other, less costly techniques have been proposed [52, 118]. One approach is to have n disks in an array, with one disk serving as the "parity" or "checksum" disk, which allows the system to recover from the loss of any single disk using the data from the $n - 1$ remaining disks. This and other configurations that allow the system to recover from multiple disk failures or to provide variable price/performance characteristics [25, 101] are proposed by the RAID project.

2.3.5 File Layout

The issue of optimizing file placement is relatively simple on a single-disk file system, but file systems spanning multiple disks open new opportunities. There are three basic and independent approaches: balancing device load, clustering active data in the center of

the disk, and striping files. The first approach attempts to prevent any single disk from becoming a bottleneck, the second attempts to minimize disk head movement, and the third attempts to maximize throughput for large files. The second approach, data clustering, is discussed in more detail in Section 2.5.

For computers with a single type of secondary storage, load balancing is well understood [120]. By measuring past file activity, the system attempts to balance the total activity for each device by moving files from disk to disk. Load balancing is generally not done in UNIX systems. In the near future, with the advent of (fast) electronic disks and (slow) optical jukeboxes, the potential benefits of load balancing across these various devices (balanced relative to device performance) will increase dramatically; systems should be able to realize a performance similar to the expensive fast devices at a cost similar to the inexpensive slow devices.

The motivation for clustering active data in the center of the disk is that disk arm movement costs time. Reducing the average seek distance reduces the average access time. Similar to load balancing, this method requires that file-access patterns remain stable over long periods of time, e.g., one or more days.

Disk striping increases file throughput by spreading a file across several disks, so that accesses may proceed on all disks in parallel. This technique is very successful only for sequential scans of large files. However, most active files are small [119]. In addition, overall system throughput and response time might be degraded by striping small files, since reading small, striped files might cost two or more seek and rotational delays instead of just one such delay. Designs incorporating disk striping must avoid such pitfalls.

A simple and effective technique that can take advantage of the benefits of load balancing and disk striping is disk interleaving. In this scheme, a file system contains n disks. Each disk k contains the blocks i for which $k = i \bmod n$. Files are striped across several disks automatically, and disk load should be roughly balanced[1]. This approach is used in the Plan 9 file system [92]. However, this approach has the same problems with small files as does disk striping.

[1]In UNIX, it appears that the most active data are the inodes, which will be uniformly distributed over the disks using this method. Consequently, most of the activity will also be "uniformly" distributed.

- Maximum data transfer rate
  - when data present in cache
  - when data on disk
- Time to complete a given set of tasks
- Fixed Load Profile
  - Number of I/O's completed in a fixed time interval
  - Number of bytes transferred in a given time interval

Table 2: Performance Measures

2.4 File System Performance Measures

There are several programs that attempt to provide some yardstick by which file system performance may be measured. Most of these benchmarks measure one or both of maximum file transfer rate (from cache or from disk) and average request latency. Some try to provide more realistic measures of system performance by timing the execution of an I/O-intensive script.

There are many possible measures of system performance, and a good benchmark may well use more than one to describe a system's capabilities and limitations. Table 2 gives a taxonomy for the various benchmarks.

All of the existing benchmarks share one or more of the following deficiencies: they access a small number of files, measure system performance over a relatively short period of time, and do not reflect the fact that file-access patterns are skewed, and most files live a very short time (in UNIX, at least).

2.4.1 Maximum Transfer Rate

There are several benchmarks that attempt to measure the maximum transfer rate to/from the disk through the file system. In general, these benchmarks attempt to measure the maximum throughput to/from the disk, and they attempt to eliminate the effects of the buffer cache on file system performance. Typically, they create a single large file and then measure the time required to read and write the file. They attempt to reduce or eliminate the impact of the disk buffer cache by creating a file larger than the buffer size.

Two such benchmarks are IOStone and Bonnie. IOStone is a synthetic benchmark that uses a single file and measures the time to read varying amounts of data from that file [86]. Bonnie and its predecessor FSX create a large file and measure the time to read, write, and

1. create many directories and sub-directories
2. transfer lots of data by copying files into new directories
3. check ownership and permissions for each file
4. read lots of data by reading each file
5. compile several programs in directory (temporary compiler files are *not* in the test directory)

Table 3: Andrew Phases

modify data in the file. The default size of the test file is 100 Mbytes, but this size can be selected by the user.

2.4.2 Task Completion Time

This style of benchmark attempts to measure file system performance at the user level by measuring the time required to complete a set of tasks. The most widely distributed benchmark of this form is the Andrew Benchmark [67].

The advantage of this type of benchmark is that it can measure the time to execute real user tasks. There are two styles of benchmark that fall into this category. The first is characterized by an artificial script that emulates "sample" user behavior, but this type of benchmark is only as good as the task selection. If the benchmark designer chooses an unrepresentative set of tasks, then the benchmark will not reflect real user activity. The second style of benchmark is based on traces of actual system usage. Unfortunately, there are no "standard" benchmarks of this type.

The Andrew Benchmark uses a script to model user behavior. It seems to be the most realistic benchmark for modeling short-term file system performance. The script has five phases as shown in Table 3, each of which attempts to emulate a different pattern of file system access. The benchmark measures the time to complete each phase.

2.4.3 Fixed Load Profile

This class of benchmarks measures file system performance under a given load. It usually measures either the number of operations completed during a given time or the amount of data transferred during a given time.

Nhfsstone measures NFS file server performance. It measures the average time required

for each of the 17 NFS operations. It attempts to force the NFS client to generate a given profile of NFS operations at a given rate. It can report both the overall average response time and the average response time for each type of NFS operation. Since the different NFS operations have different average response times, the overall average response time is very sensitive to the mix of NFS operations executed by the benchmark.

IOBench measures transaction performance [121]. It models system load by varying the concurrency level, which is the number of jobs generating requests. It also varies the processing "complexity" by varying the amount of CPU work done by each job, and it varies the I/O "complexity" by varying the number and type of file accesses done by each job. It also divides the work into "transactions," which consist of fixed amounts of processing and I/O. Each job executes as many transactions as possible in the given time. IOBench outputs the CPU utilization, the amount of data transferred, and the number of transactions completed.

2.5 Data Clustering

One useful optimization, which can be done automatically, is clustering active files close to each other and in the center of the disk, with less active files placed on either side of them. This is known as the "organ pipe" optimization. The hope is that this organization will reduce the seek time between accesses. If it is likely that consecutive accesses are to active files, then it is likely that they will be to blocks in the center of the disk, reducing the expected seek time. The rotational delays may also be reduced, if it is likelier that accesses will be to the same cylinder.

Placing active data near the center of a disk is not a new idea. In 1973, it was shown that placement of active data in the center of the disk is optimal [58, 122], but how this optimal placement would be achieved in real systems was not discussed. For years, some system administrators have been placing a few of their hottest files in the center manually [9]. iPcress automates this process, and Chapter 7 reports the performance gains. Such reorganization will be done frequently, probably nightly, and possibly even continuously.

Vongsathorn [117] proposes to add functionality to the UNIX device driver to allow it to reorganize the disk according to cylinder access probabilities in an organ pipe fashion, which is a good first approach to the problem of reorganizing data. However, the granularity, 1 cylinder, seems too large. Similar work has been done by Ruemmler [95], except that he

evaluated the benefits of reorganizing the disk in terms of blocks, tracks, half-cylinders, and cylinders. Vongsathorn did not quantify the effectiveness of clustering, but Ruemmler observes speedups between -35 and 23% in average access time. Ruemmler also found that reorganizing the disk works best when the disk receives mostly read activity. The negative speedups might be due to their method of reorganization, which separates logical blocks that the file system perceives as adjacent, so UNIX's block allocation strategy allocates adjacent logical blocks for a file but the physical blocks are widely separated.

Disk Express II [30] monitors disk use on a per-file basis and reorganizes the disk nightly based on the access history. It places the most active data close to cylinder 0 (the inner-most track of the disk) and dormant data near the outer-most cylinder, leaving all the free space in the middle. It places the active data near the beginning of the disk because the MacIntosh file table is located there and cannot be moved. Disk Express II is a complete system, but there are no reports of its effectiveness in the open literature.

Chapter 3

File Access Patterns

In order to design a high-performance file system, we must first understand how it is likely to be used. Knowing typical access patterns might help file-system designers improve system performance, especially patterns of file-system use that could predict future behavior.

3.1 File Temperature

A small fraction of files absorbs most of the activity, so improving performance for these files can improve system performance. However, in order to handle these files effectively, we must identify them and how they are used.

For caches, the terms *hot* and *cold* denote objects that are accessed heavily and lightly, respectively. For files, the number of I/O's to a file approximates its frequency of access. But not all files have the same size, so this approximation is inadequate. For example, if two files each receive a hundred 512-byte I/O's, and one file is a 1,000 bytes long while the other is a 1,000,000 bytes long, then the 1,000 byte file is obviously "hotter." We define *file temperature* as the number of bytes transferred divided by the length of the file. The two files from the example above have vastly different temperatures.

3.2 Trace Data

We conducted experiments to determine if file-access patterns could predict future behavior. Amdahl Corporation provided trace data and computing resources; detailed results may be found in Reference [110]. We were primarily interested in dynamic access patterns. The

- which file was opened
- roughly how many I/O's were done during the session
- file open mode (read only, write only, read/write)
- when the file was opened
- when the file was closed
- how many tracks were allocated to the file when closed
- name of the job that opened the file
- logical name assigned to the file by the job

Table 4: SMF File Open Session Information

data was generated by IBM's System Management Facility (SMF), which provided trace data for each file open and close in IBM's MVS/XA operating system.

The data came from two installations, which we call customer "S" and customer "Z." The only information we received was SMF trace data. The first trace, from customer "S," contained a week's worth of data and 170,437 open-close sessions. The second trace contained three days' data with 142,795 sessions. Most of the analysis was done using SAS and Merrill's MXG Package.[1] Essentially, SMF records file activity information for each active process and open file. This data is transformed into SAS data sets by Merrill's MXG package.

The SMF traces contain tremendous quantities of data, most of which was irrelevant; we extracted a table of file open-close sessions. Table 4 summarizes the information recorded for each session.

The SMF traces had both limitations and errors in the data. However, we desired data from large systems that were being used in a "production" environment, and we could not modify and install an operating system that could provide more accurate or detailed information.

SMF data is somewhat unreliable; SMF sometimes records invalid data, but these

[1]SAS is a statistical analysis product, and MXG is a set of SAS macros.

records can be ignored. In particular, activity measurement facility for VSAM[2] files incorrectly records activity to files that are being accessed concurrently by more than one process. Consequently, our results are for non-VSAM files.

SMF data cannot be used to compute file temperatures. Since SMF records only the number of I/O's and the number of tracks allocated, we use a working definition of file temperature as the number of I/O's over the number of tracks allocated. There is no record of the size of these I/O's, but we assume that they are usually one track in size.

The most glaring shortcoming of SMF is that it collects only summary statistics for each open. There is no record of the timing between various reads or which blocks or tracks were accessed. In addition, SMF merely counts the number of I/O's, without distinguishing between reads and writes for files opened in read/write mode.

SMF records the number of tracks allocated to a file not the actual space used. MVS users specify the initial size for a file at file creation time. This size is the space allocated for the file, but the true file size is the number of bytes of data stored. The difference between the allocated and true sizes causes some experimental error in our temperature calculations. In addition, small files often have allocations of one cylinder or a few hundred thousand bytes,[3] and the smallest unit of allocation is one track (13–47 Kbytes), so the error is greatest for small files. These errors tend to lower file temperatures.

Another problem with the SMF data is that we have data for only those files accessed during the measurement period. We cannot tell how large the entire file system is, nor can we tell when the last access to each file was prior to the trace period.

In MVS, a partitioned data set is a file that consists of a set of member files. It is impossible to tell from the SMF data which members of a partitioned data set were accessed on a given open; SMF records only the name of the partitioned data set. Essentially, logically separate files are being treated as a single unit. As a consequence, the temperature of the group is the average of the individual temperatures, and the temperature for each session is computed as if each member had been opened. This effect dilutes the temperature of the hottest members producing a uniform lukewarm temperature. Since some members are probably hotter than others, this dilution tends to squelch hot spots.

[2]Virtual Sequential Access Method. VSAM files have a record structure and are generally used by databases.

[3]MVS users can define the default allocation for new files. Since files growing beyond the initial allocation may require user intervention, users tend to over-estimate the allocation size and set the default allocation size to one cylinder.

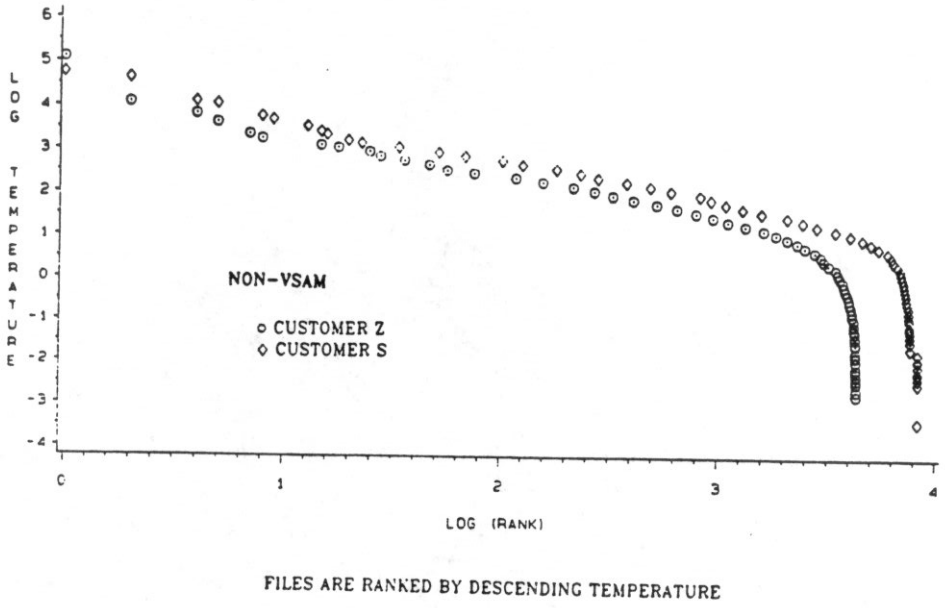


Figure 1: File Temperature

One interesting feature of the data is that there are thousands of temporary files that consume 10–25% of the total I/O's and whose cumulative size is 50–80% of the total space consumed by all files. However, the space used by temporary files at any given time is relatively small, which produces an interesting accounting problem from the standpoint of file system size; there is no easy way to account for the space used by temporary files. The problem is that the space is reused by the system, and it is not obvious how to attribute the I/O's to particular locations. As a result, we ignored temporary files assuming that they require little space.

3.3 Analysis Results

Figure 1 shows that long-term file temperatures are strongly skewed; a handful of files are very hot, but most files are cold. Figure 1 is a log-log plot of file rank versus file temperature for both customers "S" and "Z." File rank is assigned according to descending file temperature with the hottest file having rank 1. Note that file temperatures varied by over five orders of magnitude.

In order to detect a correlation between some criterion, such as file temperature or the number of open-close sessions, and file activity, we rank the files according to that

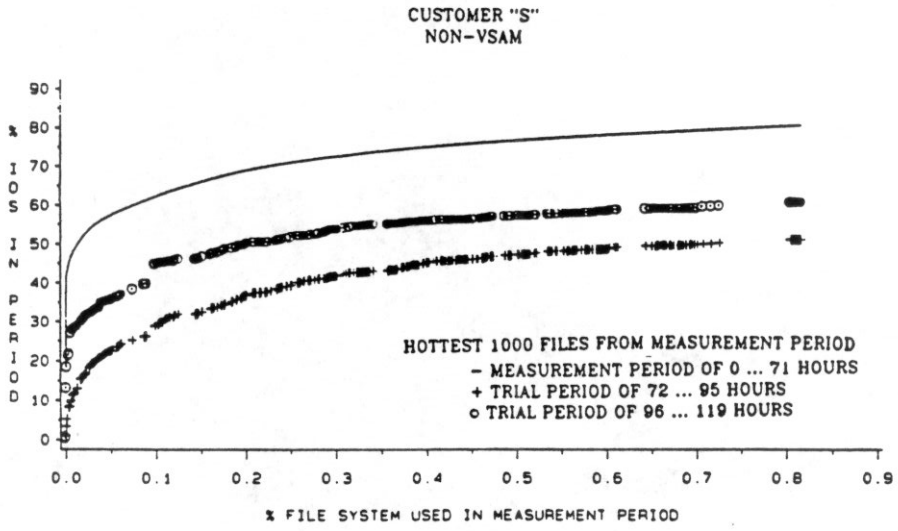


Figure 2: File Temperature Persistence

criterion and then plot the cumulative I/O versus the cumulative disk space consumption both expressed as percentages.

The shape of the resulting curve indicates whether the criterion used to rank the files is correlated with file activity. For example, a curve that starts at the origin, heads nearly straight up to the top of the graph, flattens out and then goes across to the upper right indicates that the criterion is highly correlated with file activity. The curves in Figure 2 are examples; they demonstrate that 0.1% of the total space used by the file system receives 30–60% of the I/O activity. Conversely, a curve that goes almost straight from the origin to the top right corner indicates minimal correlation with file activity. The curves in Figure 5 are examples; 10% of the file system disk space receives 10–20% of the I/O activity.

The hottest files tend to stay hot for days. We can show this effect by measuring file temperature over a sample period, say 1–3 days, and ordering the files by descending temperature. We can then measure the cumulative I/O over that *same* ranking for the next few days to see if the the hottest files still get most of the I/O. Figure 2 shows the cumulative percent of I/O versus cumulative percent of the file system for a three-day measuring period and two subsequent days; only the hottest 1,000 files are shown. The I/O percentage is relative to the total I/O during each of the three periods, and the space is the total space used by all permanent files during the measurement period. It is evident that 50–60% of the I/O's are still directed towards these hot files, which suggests that current file temperature

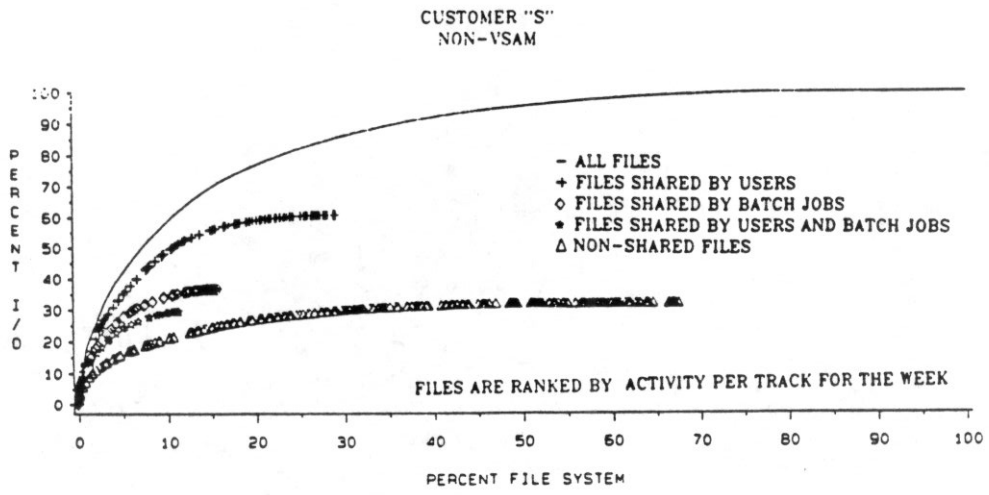


Figure 3: Analysis of File Sharing

predicts future file temperature.

Figure 3 reveals that, in general, shared files are hotter than non-shared files. Figure 3 shows the cumulative percentage of all I/O versus the cumulative space for all files, files shared by users, files shared by batch jobs, and non-shared files. The top curve represents all files (shown with a solid line), and the second curve from the top represents files that were opened by more than one user ("+" signs). In order, the remaining curves are for files used by more than one batch job (diamonds), files used by more than one user and by more than one batch job (stars), and finally files used by at most one batch job and one user (triangles). Note that it does not make sense to "add" two curves together; the set of files shared by users and the set of files shared by batch jobs are not disjoint, and within each set the files are ranked by descending temperature.

We expected that file size might be correlated with temperature, e.g., large files might tend to be shared system files, such as database files. We also expected that frequently-opened files would be hotter than other files, since they appear to be accessed frequently. However, neither file size nor number of file opens within a given interval were correlated with file temperature, and therefore neither would be good predictors of future temperature. Figures 4 and 5 show the cumulative percent of I/O and the cumulative percent of disk space in the file system for the files with ranking determined by size and number of opens,

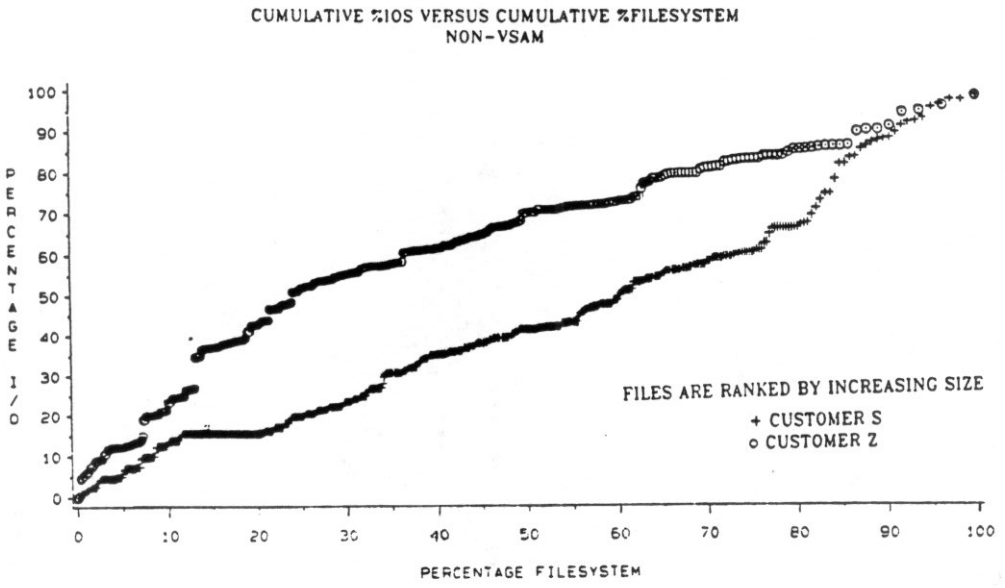


Figure 4: Files Ranked By Increasing File Size

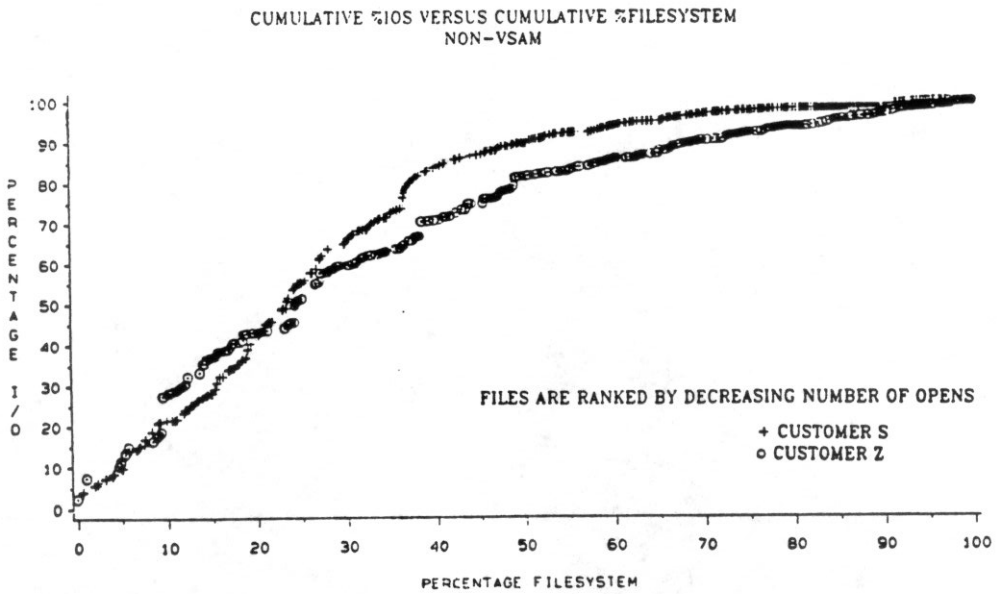


Figure 5: Files Ranked By Decreasing Number of Opens

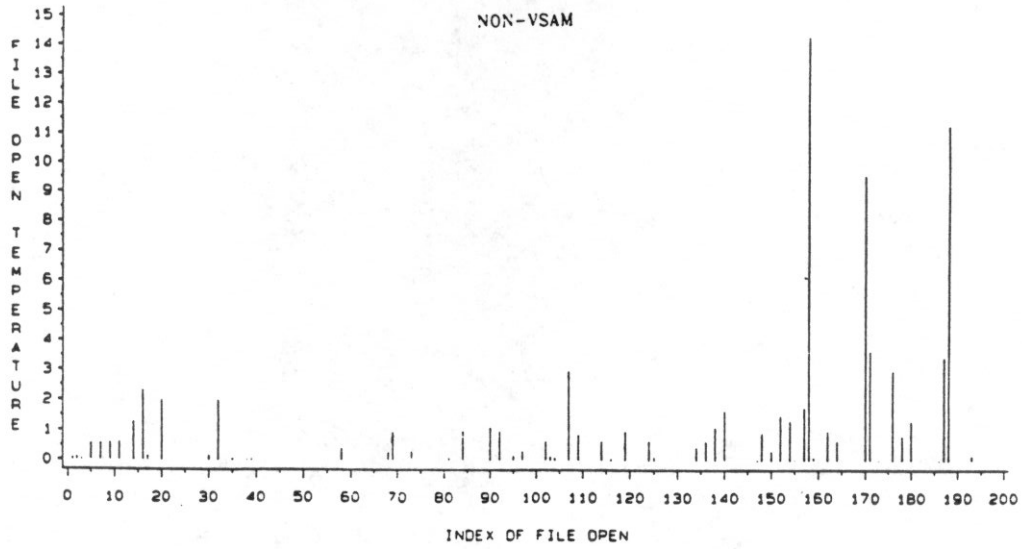


Figure 6: File Open Temperatures for Sample File

respectively.

We also tried to predict the file temperature for individual open-close sessions based on prior individual session temperatures, but, in general, file temperature during the previous open-close sessions does not predict file temperature during the next session. For example, Figure 6 shows the sequence of file temperatures during each open-close session for a sample file. Individual open-close sessions are indexed on the x axis, and the session temperatures are labelled as "file open temperature" on the y axis. For this file, most sessions have a negligible temperature, but occasionally there is an open-close session with a lot of activity. The interesting sessions are the active ones, and we would nearly always guess that the file would be inactive based on the activity during the last few sessions, so this statistic does not help predict future active sessions. It is possible that *some* files will have predictable file session temperatures, but this does not seem to be the case in general.

Since file open temperatures are not constant, the file temperature probability density function (PDF) is an important property for each file. If the PDF is "regular" in some sense, then the cache could guess more accurately which files to promote or demote. A "regular" PDF might be an exponential or uniform distribution. PDF's are not directly observable, but the histograms of file open temperatures provide reasonable approximations.

Figure 7 displays some file open temperature histograms for a sample of 21 files. This

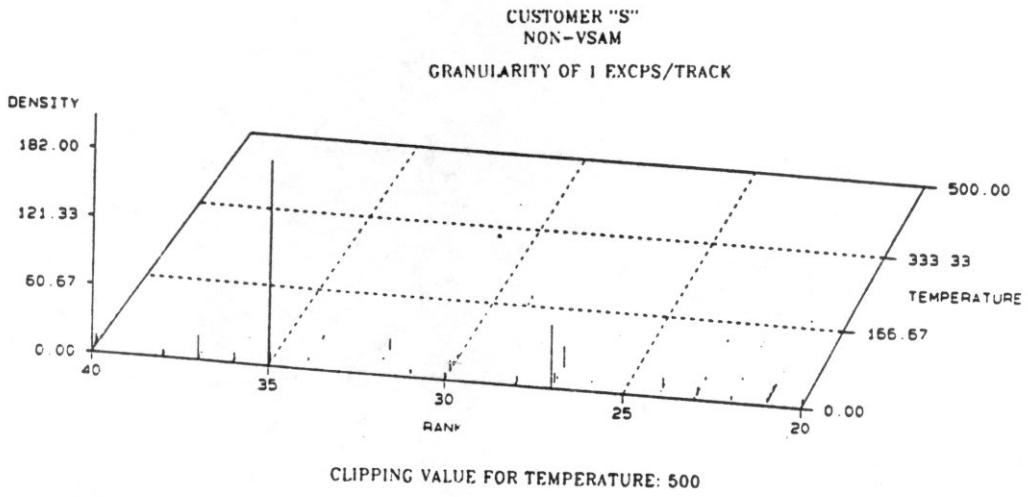


Figure 7: File Open Temperature Histograms for 21 Sample Files

figure is three dimensional, with the files arrayed on the "rank" axis, the file open temperature "buckets" on the "temperature" axis, and the number of opens per "bucket" on the "density" axis. The "buckets" are the conversion of the real-valued file open temperatures to a discrete axis. The rank is identical to the rank in Figure 1, and all temperature values associated with a given file have the same rank. Many file open temperature histograms seem to be random scatter plots rather than regular functions.

An interesting file property is the PDF for the time from a file's close to its next open, which could be used by the cache replacement strategy to flush files with a lower probability of being re-opened. Again, we approximated these PDF's with histograms. Some files exhibit exponential-like distributions, whereas others exhibit more uniform distibutions. Storing some form of the PDF for the time to next open by storing a small histogram would permit the replacement strategy to compute the "probability of being accessed within the next n seconds" based on this histogram and the time of last access. The system could use this "probability" to rank the files every few minutes and use this ranking as a hint when flushing the cache.

Figure 8 is a three dimensional graph, similar to Figure 7, with axes "rank," "seconds," and "density." The axes "rank" and "density" have meanings similar to the axes in Figure 7. The axis "seconds" represents the number of seconds between each file close and the next

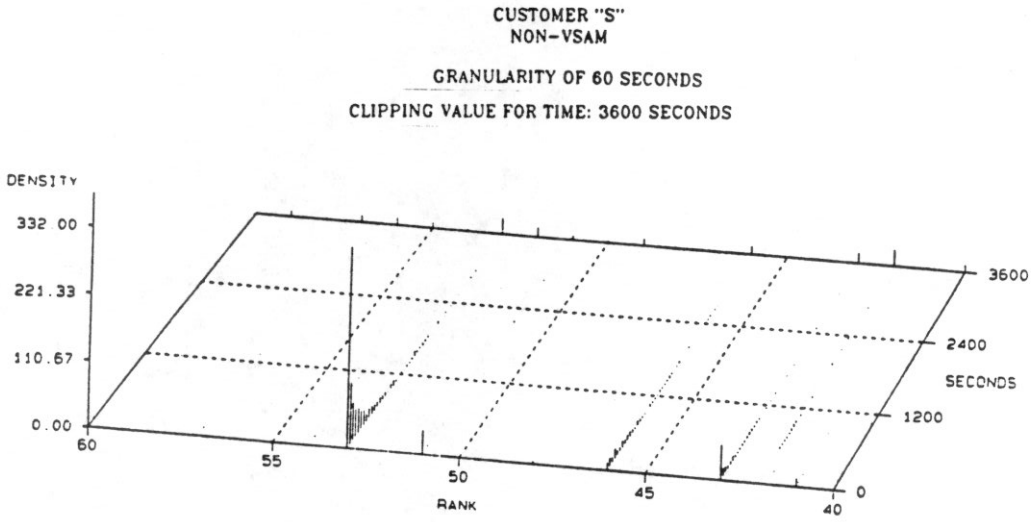


Figure 8: Histograms of Time Between Last Close and Next Open

open. Negative values imply that the file was reopened before it had been closed. Rather than amalgamate overlapping open-close sessions into a single, larger session, we discarded the negative values, since they were rare. In addition, the "seconds" axis has been clipped at one hour, and the density at the time one hour is really the density for all time greater than one hour.

Some files seem to have recognizable, or at least reasonable, PDF's for the time from last close to next open. For example, files 43 and 53 seem to have exponential-like distributions. Unfortunately, other files, such as 51 and 52, exhibit more complex behavior.

3.4 Extrapolating to UNIX

We would like to assume that the patterns we found in MVS also occur in UNIX. We could not repeat our experiments on traces from UNIX machines because we did not have such traces. However, verifying the existence of stable hot spots can be accomplished without detailed traces.

Other sources suggest that UNIX also has stable hot spots. For example, a recent study of nine UNIX machines found that, on average, only 1% of the 75% coldest data space is accessed on any given day [21]. This result implies that 25% of the data gets all the

remaining activity. In addition, most accesses are to files that have been accessed recently.

Most opens are for files opened hundreds or thousands of times a week [42]. By itself, this result does not mean that there are stable hot spots since most of those open-close sessions may not transfer any data, but it is indicative. In addition, 66% of all files had not been accessed in over a month.

3.5 Results

MVS has hot spots, and they last for days. Similar patterns appear to occur on UNIX systems, suggesting that UNIX has stable hot spots. These results imply that we can predict that active files remain active and dormant files remain dormant, and that most files are dormant. We can optimize layout and access for those files that will be very active in the near future.

Chapter 4

Design

iPcress is designed to be both a production file system, where performance and reliability are paramount, and a test-bed file system, where flexibility and modularity are vital. It is intended for large, general-purpose computer with a large memory and many disks. It is also extensible so that new techniques and algorithms may be added in the future.

iPcress is designed for both performance and reliability, but only the performance features have been implemented. The details of the implementation are discussed in Chapter 5.

We assume the hardware environment has a large memory for disk caching and multiple storage devices. Multiple storage device types, such as electronic disks, magnetic disks, and optical jukeboxes, are supported. In addition, we assume that storage device response time and throughput are the primary performance bottlenecks.

File systems should span multiple devices for reasons that include enhanced performance and reliability. A single file system spanning multiple disks can provide improved performance over several file systems that each span a single disk in several ways. It can provide better throughput by striping files, it can balance the average disk load over the devices, and it can prevent any single device from becoming a bottleneck. Multiple-device file systems can also improve file system reliability by storing enough redundant data to recover from even catastrophic device failure. Multiple-device file systems also allow the creation of files that are larger than any single device.

iPcress is a user-level server process. User requests are sent to the kernel and are forwarded to the server process; replies are returned on the reverse path. Data can be transferred using shared memory in order to reduce copying. User-level servers need operating systems that provide threads and message passing. We chose MACH because it has

- Large file cache
- Historical data on each file's access pattern
- Variety of storage techniques and caching algorithms
- Variable size blocks in both memory and secondary storage
- Multiple devices within a single file system
- User-level server process with multiple threads
- Database technology for reliability and fast recovery

Table 5: Main Points of Design

these facilities [1, 11, 114].

Making the file system a user-level process is a tradeoff between programming complexity and performance. User-level servers usually reduce development effort and always avoid operating system kernel modifications. They do increase CPU costs, but CPU costs can be largely ignored because of increased CPU speed and because the servers can always be moved into the kernel if the tradeoff warrants the effort.

Everything is a file in iPcress. File-system data structures, such as the inode table or disk block free lists, are contained in files just like user data. Also, all performance and reliability features apply equally to both user data and system data.

There is no single means of storing or accessing file data, but rather there is usually a method that is best for each "style" of file access. Most files are usually accessed using just one access style, so iPcress uses layout and caching methods for each file based on past access histories. Emphasis focusses on identifying *which* layout and caching methods are appropriate for each file. As new layout or caching methods are added to iPcress, the primary emphasis is on their automatic invocation when they are appropriate.

The basic design points of iPcress are shown in Table 5. The points that are unique to iPcress are that it stores and uses file-access histories and that it provides a variety of file layout and caching methods. Also, iPcress manages multiple devices and provides global optimizations, such as automatic load balancing between devices.

The design decisions fall into three categories: performance, reliability, and flexibility. Some decisions, such as allowing the file system to span multiple devices, improve both

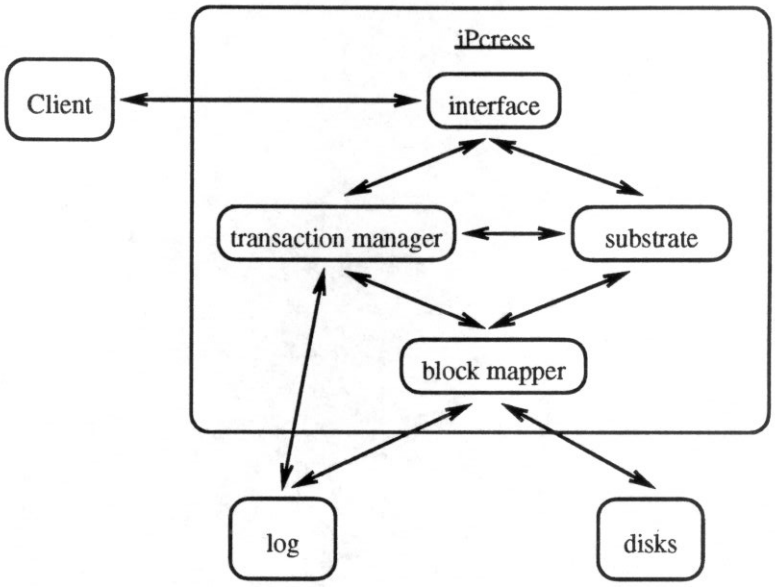


Figure 9: Block-level Diagram of iPcress

performance and reliability.

4.1 Design Overview

At the top level, iPcress has four parts: the external interface, the transaction manager, the substrate, and the block mapper. The external interface implements the external file-system semantics (for example, NFS or Sprite), and it uses the transaction manager to update file-system structures, such as directories, atomically. The substrate manages files and provides various file layout and caching algorithms. The substrate also uses the transaction manager to update file-system data structures, such as free lists, atomically. The transaction manager uses log-based recovery to provide transactions. The block mapper temporarily remaps reads from stale file data to current data in the log when the data has been written only to the log. It also manages the propagation of data from the log to the files. Figure 9 depicts these parts parts and how they interact.

The substrate has five major parts: the file cache, directory-name cache, block cache, memory manager, and disk manager. The file cache is a set of file objects that are currently *active*, i.e., they have some data cached in the system. The directory-name cache holds the most recently accessed directory entries. The block cache is an LRU cache of variable

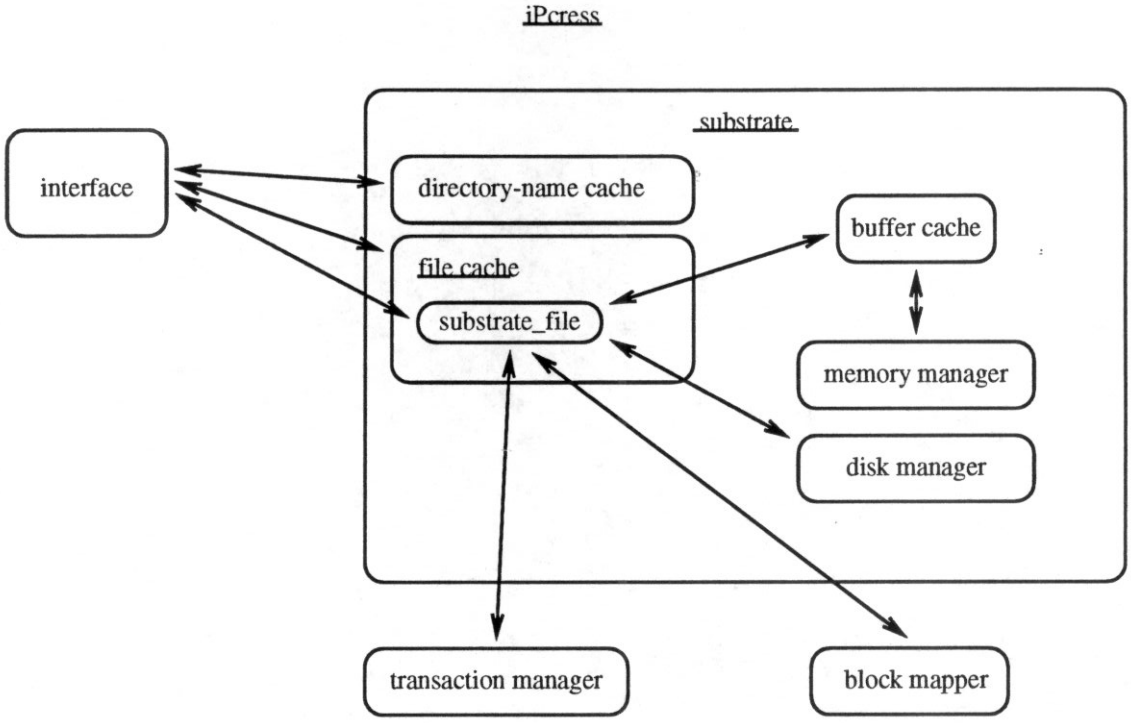


Figure 10: Block-level Diagram of the Substrate

sized buffers. The memory and disk managers control the free memory and disk space, respectively. The inter-relationship of these components is shown in Figure 10.

The substrate manages substrate files or file objects. These objects define and implement the basic file interface including operations such as read and write. Substrate files have a variety of layout and caching algorithms available. In the object-oriented sense, operation requests are handled by the file objects themselves. File objects are discussed in more detail in Section 4.5.

Following the progress of a read request illustrates how the system works. When a read request is sent to iPcress, the external interface accepts it and begins processing it. If the request accesses an existing file, the external interface looks the file up in the file cache, and if it is found, the file object handles the request. If there is a cache miss, the file cache creates an active version of the file to handle the request. When necessary, the file object requests buffer space from the buffer cache, which in turn may request space from the memory manager. When the file object completes the request, the external interface sends a reply to the client.

4.2 Performance

iPcress uses a large cache to reduce disk accesses. Caching decisions are split into staging, which moves data from disk to cache, and flushing, which moves data from cache to disk. The cache has two parts, a file cache and a block cache. The file cache holds file objects, which are responsible for deciding which data should be staged from disk into the block cache. The block cache is an LRU cache that decides which data should be flushed. Using a file cache allows us to vary caching strategies on a per-file basis, and to use predictive methods for staging. For example, small files can be staged completely when they are opened, while large files can use a complex strategy to stage blocks predictively.

iPcress maintains an access history for each file. This information is necessary for any type of automated optimization. A system for clustering active data in the center of disk is described in more detail in Section 7.5. Other possible optimizations include dynamic device load balancing and automatic replication of active read-only files.

Variable sized blocks are used to avoid some of the high overhead associated with each I/O, which is much higher than the cost of accessing additional blocks within a single I/O. The completion time for an I/O is $T_{I/O} = T_{\text{overhead}} + n_{\text{blocks}} T_{\text{block transfer}}$. Experimentally, we found that $T_{\text{overhead}} \approx 100 T_{\text{block transfer}}$ on a VAX 11/785 running MACH with a UDA50 controller and an RA81 disk, so minimizing the number of I/O's to access a file can greatly improve system performance.

The log helps improve the system's write performance. Since files tend to be either read-only or write-only, and since most UNIX files have short lifetimes [85], we use the log to hold recently written data. Log-record forwarding and data copying from the log to the file helps free space at the tail of the circular log. We expect that most log records will not require either forwarding or copying because the data in the records will have been deleted or rewritten [17, 94].

The block mapper manages read access to blocks that have been written to the log but not yet propagated to disk. Since the log is on disk, the block mapper redirects read accesses from the data's permanent location, which contains stale data, to the data's current location in the log. This approach might provide write performance similar to that of the log-structured file system [94]. Since the log is circular, valid data that is about to be overwritten must be forwarded to the head of the log or copied to its permanent position.

4.3 Reliability

Reliability encompasses recovery from interrupted operation and media failure. Interrupted operations might be incomplete and might leave the file system in an inconsistent state. For media failures, some portion of a device is no longer accessible, such as a scratched disk surface caused by a head crash.

Since iPcress stores all file-system data in files, the reliability mechanism operates on files. The reliability mechanisms for user data are optional because of the storage and performance overhead involved.

Recovering from interrupted operation involves eliminating the effects of incomplete operations. Database systems use transactions and logging to recover from these failures. However, database transactions usually involve the modification of fields within records of various relations. File-system operations, particularly on UNIX, simply modify byte ranges within files. Consequently, we developed the analogous concept of file-system transactions, which atomically update byte ranges within one or more files using log-based recovery techniques. The log is a circular log and it is stored on disk for performance reasons.

With media failure, all data is at risk regardless of when it was last accessed. The system must keep redundant information so that it can reconstruct the original image from the remaining pieces of the data. Not all data is equally important and there are tradeoffs between extra storage used for redundant data and data availability. Therefore, we define reliability as a per-file property.

iPcress's flexibility provides a range of options. The simplest method is mirroring or shadowing, in which there are two or more copies of the file on separate disks. The advantages of this method are that the other copy is immediately available, and that read performance may be improved because the disk with the fastest response time can be read. The disadvantage is the space required for the copy.

For large files, more space-efficient methods may be used. For example, if the file system contains n disks, the file data may be striped over $n-1$ of the disks while the n^{th} disk contains the exclusive-OR of the other $n-1$ data blocks. When any single disk fails, the file data contained on that disk may be recovered by exclusive-OR'ing the remaining $n-1$ data blocks. These and other techniques have been explored more fully in the RAID project, and most of the techniques described in References [50, 51, 52] could be used on a per-file basis in iPcress.

4.4 Flexibility

iPcress is intended as a test-bed, and the design is modular and avoids specific algorithms whenever possible. For example, there are no mandated file layout and caching algorithms, or buffer cache replacement policies. Instead, we specify only the interfaces between modules in iPcress, hiding the implementation details. One major exception to this rule is the use of variable size blocks with a buddy system allocation scheme.

We use functional interfaces between program modules wherever possible, and minimize state information within each module that is accessible to the other modules. This approach has permitted large modifications to central modules (e.g., changing the buffer cache replacement policy) with no change to other modules.

4.5 Design Details

The most important object in iPcress is the file object, which provides the basic file operations. There are many different types of file objects that share a common functional interface, which provides the same basic operations, such as read or write. In addition, each file object type can be described by two or more *properties*. The two basic properties are storage method and caching method. There are several variants of these methods, and others can be added. File objects can change these methods as necessary. In addition, each file object tracks its own access history.

The storage property determines how a file is stored on disk. For example, small files may keep their data in the inode, while large files may be kept in multiple blocks on several devices. It is also possible to have a file stored reliably, by using redundant data spread over several devices. The caching property determines how the file is cached and it stages buffers from secondary storage into the cache. There are many techniques available. Some files might be completely staged when they are opened, others might use more sophisticated techniques. Each property can be viewed as a complete virtual file interface similar to a vnode interface [75]. In particular, the cache and storage properties are layered interfaces.

File-system data structures are kept in a hierarchy of files as shown in Figure 12. The root is the super inode file, and it has a fixed size and lives in a fixed location on disk. This super-inode file contains the inodes for five key system files: the UNIX inode file, the disk-block free-list inode file, the UNIX inode free list, the log file, and the super log file. The UNIX inode file contains the inode table for all user files, and the inode free list contains a

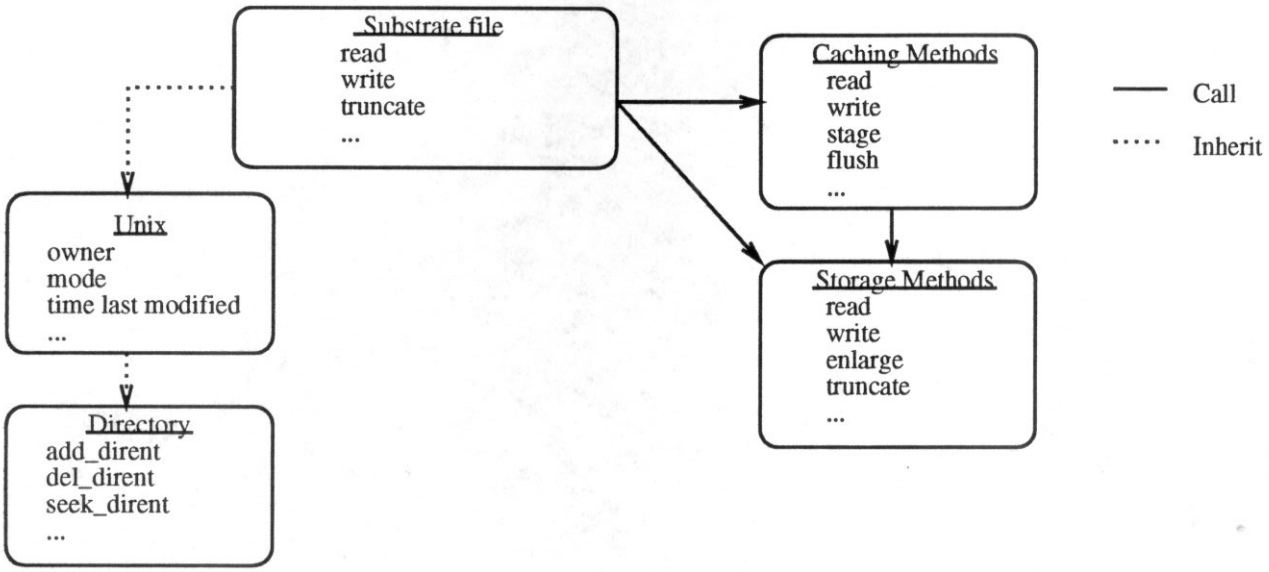


Figure 11: Block-level Diagram of File Objects

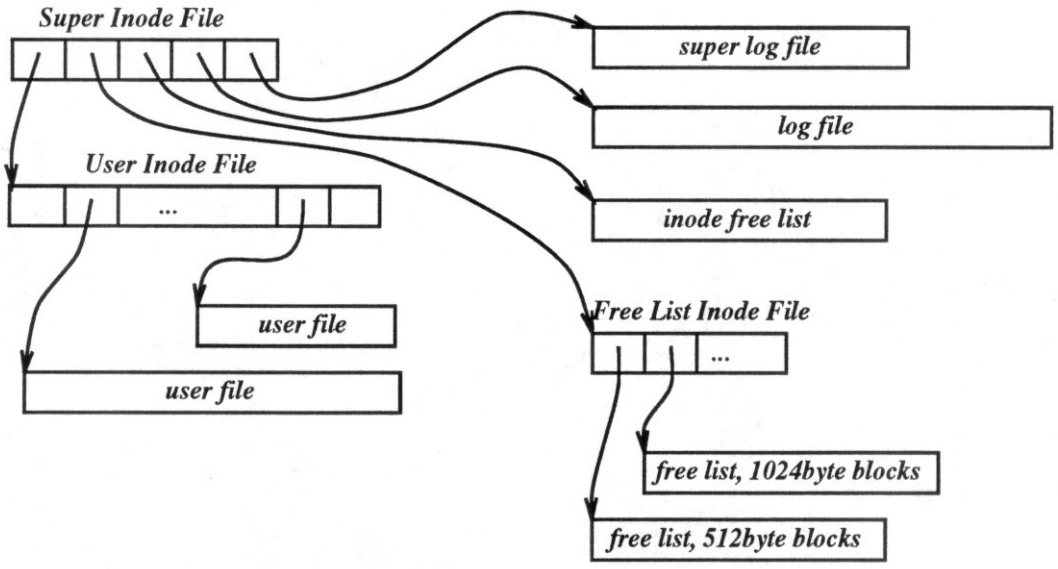


Figure 12: File System Data Structures

list of free inodes in the UNIX inode table. The free-list inode file contains inodes for the disk-block free lists, each of which contains a list of all free blocks of a given size. The log file is used by the transaction manager to log modifications to all files except those that alter the size or placement of the log file itself, and those modifications are logged in the super log file whose size and location are fixed at file system creation.

Each file is identified by its internal address. File addresses are a list of the inode indexes between the root and the file. For example, the user inode file has address $\langle 0 \rangle$, and the 1024-byte block free list has address $\langle 1, 1 \rangle$. User files have addresses of the form $\langle 0, k \rangle$.

Chapter 5

Implementation

iPcress is implemented as a user-level NFS server. It runs on a DECstation 3100 with RZ55 (330 Mbytes) and RZ56 (660 Mbytes) disks. iPcress is a user-level NFS server because NFS is a standard interface used by many systems, and there are implementations of the NFS interface for user-level servers. As a result, iPcress did not have to be part of the kernel, the interface code was borrowed from other sources, and all UNIX programs can access iPcress because it uses a standard interface. The initial development was on a VAX under MACH, but was completed on a DECstation 3100 under ULTRIX.

5.1 NFS interface

Currently, NFS is the only external interface implemented in iPcress. This interface is simple and well defined and has a small set of operations. Conceptually, there are two types of machines, servers and clients. Servers provide file service to clients. A machine may be both a client and a server. iPcress exports its files to clients.

The NFS interface has 17 operations and is supposed to be stateless, which means that operations like `read` or `write` are supposed to be idempotent. However, some operations, such as `create` and `remove`, are not.

NFS has no explicit mechanism to handle consistency. Instead, it relies on the client to flush cached data and meta-data periodically, and to get fresh, possibly updated, data from the server. Most clients use a 30-second age limit on data.

iPcress's implementation of the NFS interface was provided by Alex Siegel of Cornell University and is a modified version of the NFS interface used in the Deceit File System [103].

Operation	Description
`type`	file type, e.g., regular or directory
`mode`	UNIX permission bits
`nlink`	number of links to file
`uid`	owner's user identification
`gid`	file's group identification
`size`	size of file in bytes
`blocksize`	file system block size
`rdev`	file system device identification
`blocks`	number of blocks used by file
`fsid`	file system identification
`fileid`	unique (per file system) file identification
`atime`	last file access timestamp
`mtime`	last file modification timestamp
`ctime`	file creation timestamp

Table 6: NFS File Attribute Structure

The interface also provided much of the code necessary to implement UNIX semantics for the various NFS operations.

NFS uses a small set of data structures. The *file handle* is a 32-byte array that identifies each file on a server and is like a capability. File handles specify the file (or directory) in NFS requests and replies. *File attributes* describe the properties of a file, such as the size and ownership of the file. Table 6 is a complete list of file attribute properties.

The full list of NFS operations is shown in Table 7. Also, Table 10 in Chapter 6 lists typical operation frequencies, taken from Legato's Nhfsstone NFS benchmark. We used the operation frequency information to decide the relative importance of each operation.

iPcress caches file handles in order to improve the performance of the `lookup`, `read`, `write`, and `getattr` operations, all of which return file attributes. The file handle cache is indexed by the NFS file handle and contains data on each file including its NFS attributes. The file handle cache is used by NFS interface to speedup `lookup` and `getattr`, which account for nearly half of the NFS traffic. The file handle cache uses demand fetching and LRU replacement.

Operation	Description
`lookup`	given a directory (file handle) and a name (string), return a file (file handle) or error
`read`	given a file and a byte range, return the appropriate bytes from the file
`write`	given a file, a byte range, and data, write the data into the file, extending the file if necessary
`create`	given a directory, a name, and file attributes, create a file in the directory with name, returning the new file
`remove`	given a directory and a name, remove the file from the directory, destroying the file if necessary
`rename`	given the old directory, the old name, the new directory, and the new name, rename the file so it has the new name in the new directory
`mkdir`	given a directory, a name, and file attributes, create a directory in the given directory with name, returning the new file
`rmdir`	given a directory and a name, remove (and destroy) the child directory from the parent directory
`getattr`	given a file, return the ownership and permissions information
`setattr`	given a file and the file attributes, change the file attributes accordingly
`link`	given a file, a directory and a name, add a new (named) link to file from directory
`symlink`	given a directory, a name and a target name create a symbolic link in directory to target file from name
`readlink`	given a directory and a name return the target name of the symbolic link
`readdir`	given a directory, a magic cookie (32-byte array) and a size, return as many directory entries as will fit in size bytes using the magic cookie as an offset into the directory
`root`	return the file handle of the root directory of the file system
`statfs`	return the status of the file system (size, bytes used, bytes free, etc.)
`null`	no operation

Table 7: NFS Operations

5.2 Substrate

The substrate provides the basic file services both for the meta-data and for the UNIX/NFS interface. Within the substrate, data is managed in terms of files. Files, in turn, store data in blocks, which may be in memory or on disk. Files usually request memory blocks from the buffer cache, and the buffer cache may request files to release blocks so they may be replaced. In addition, there is a maximum size limit for buffers, which is currently 32 Kbytes.

The file cache contains active file objects. Unlike conventional, fixed-size caches, the file cache contains all active file objects and limits its size by de-activating those active files that no longer have any data contained in the cache. As inactive files are accessed and activated they are added to the file cache. Files are addressed in the file cache by their internal address.

File objects have two properties: cache type and storage type. These types implement the caching and storage methods described in Chapter 4. Files may also have UNIX and directory attributes. UNIX file objects inherit all of the operations and attributes of substrate files, including the independent caching and storage types. UNIX file objects also have attributes such as ownership and access permissions, which are required by the UNIX/NFS interface. All user data are contained in UNIX files, and only UNIX files are accessible to users. Directory file objects are UNIX file objects with the functionality of UNIX directories.

5.2.1 Cache Types

There are four cache types: immediate, whole, demand, and partial. Immediate files store all of the file data in the file's inode, and they forward data requests to the inode file. Consequently, immediate is both a caching and a storage type. Storing file data in the inode is both time- and space-efficient. Inodes are 512 bytes long and can accommodate up to 450 bytes of data.

Whole file caching is used for files that have a single disk block no larger than the maximum buffer size. Since most files are accessed sequentially and completely, this cache type minimizes the overhead of accessing all of a small file.

The demand and partial cache types are demand-based, fixed-block size caching strategies. The demand type uses 512 byte buffers and is used for inode files. The partial type uses buffers with the maximum buffer size (currently 32 Kbytes) and is used for all other

Field	Description
`mode`	access permissions
`nlink`	number of hard links to the file
`uid`	file owner's user id
`gid`	file's group id
`atime`	time file last accessed
`ctime`	time file last created
`mtime`	time file last modified

Table 8: UNIX File Attributes

files.

5.2.2 Storage Types

There are three storage types: simple, multiple, and indirect. The storage types are accessed only through cache types, and storage types access disks directly. Since disks are accessed by blocks, storage types require that byte ranges in I/O requests fall on block boundaries.

Simple files have a single data block assigned to the file, and all file data is contained in this block. The pointer to the data block is contained in the file's inode.

Multiple files are larger than simple files and their data is contained in an array of blocks. All of the blocks except possibly the last have the maximum block size. The inode contains an array of block pointers and an integer specifying how many data blocks the file contains.

Indirect files are truly large files, where the array of block pointers is too large to fit in an inode. Indirect files have an array of one or more disk blocks, each of which contain an array of disk block pointers.

5.2.3 Attributes

UNIX attributes are required by UNIX to define file ownership, security, and other standard bookkeeping functions as detailed in Table 8. These attributes are stored in the inode.

UNIX directory file objects implement UNIX directories, and define a mapping from a name to an inode index. iPcress implements directories using hash tables with double hashing. When necessary, the directory hash table is resized to accommodate additional entries or to reclaim space when the directory shrinks.

Field	Description
table_size	size of the hash table
num_entries	number of directory entries in the hash table
num_deleted	number of entries in the table marked deleted
str_space_occupied	number of bytes occupied by names
str_space_deleted	number of bytes occupied by deleted names

Table 9: UNIX Directory File Attributes

Directory files have two sections. The first section contains the hash table and the second contains the name strings. As new entries are added to the directory, the names are appended to the string section.

Strings are hashed using a single hashing function, which returns values over the range of unsigned integers. We use double hashing [77], and generate both hash indices using the result of hashing the name (referred to as the hash value). The average search path in double hashing is much less than simple hashing.

Double hashing uses two hash values. The primary hash index is the index of the initial probe into the hash table. The secondary hash index is the offset by which the probe moves. Since the table size is always a power of 2, the secondary hash need only be an odd value between 1 and the table size minus 1 to guarantee relative primality between the table size and the secondary hash index.

The primary hash value is the hash value modulo the table size. The secondary hash index is computed by exclusive-OR'ing the hash value with the hash value shifted right 2 bits, computing the modulus of the result by the table size minus 1, and OR'ing this result with a 1. The first step provides a "random" value and the second provides an odd value less than the table size.

The hash table entries have four values: a pointer to the beginning of the name in the string section, the name's length, its hash value, and the inode index for the file. The hash value speeds string comparison; during lookups, the actual string is examined only if both the hash value and the length equal those of the search string.

Table 9 shows the additional file attributes used by directories. These attributes are used to manage the hash table and the string section and are stored in the file's inode. The last four attributes are used to determine when the directory has enough deleted entries or strings to warrant clean up.

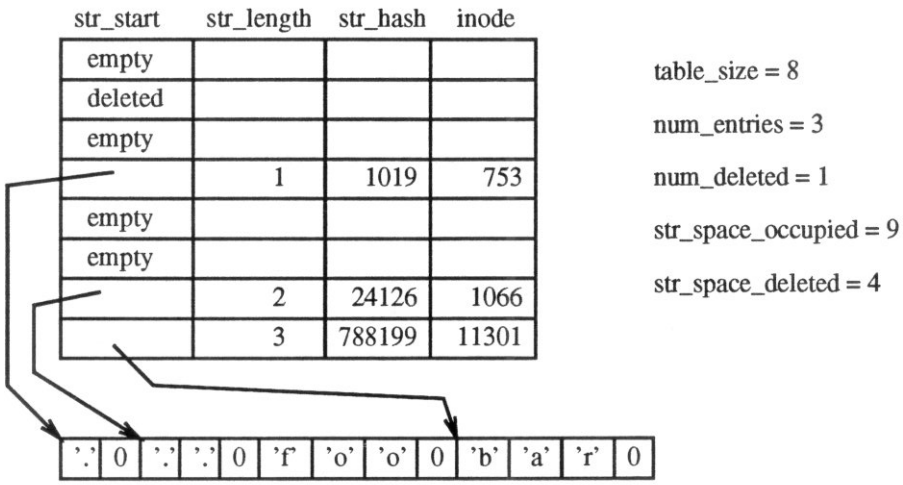


Figure 13: Directory File Structure

Figure 13 shows a sample directory file that contains three entries ".", "..", and "bar" and has had "foo" deleted. The table has three valid entries, one deleted entry, and four empty entries.

Inserting a new entry into the directory works as follows. The primary and secondary hash indices for the name string are computed and used to probe the table until either an empty or deleted slot is found. The name is appended to the file, and the hash table is updated with the appropriate values. Finally, the directory attributes are updated. Deleted entries are necessary to avoid stopping a search prematurely.

Deleting an entry from the directory is simple; the hash table entry for the file is found, the directory attributes are updated to reflect the deletion, and the entry is marked deleted.

Directories that have frequent insertions and deletions may have a large, sparsely populated string section. Consequently, the system tracks deleted space in the string section and reclaims this space when it exceeds a threshold. In addition, when the hash table is mostly empty, the system shrinks it. Finally, if the number of entries and deleted entries in the table is too large, the deleted entries are removed by reinitializing the directory and reinserting each valid entry, which also compacts the string section. This operation is potentially costly, so it is done only when necessary, or it can be done offline.

5.2.4 Buffer Cache

The buffer cache contains all file-system buffers, and its role is analogous to the UNIX buffer pool [4]. Unlike UNIX, however, the cache contains variable size buffers.

All buffers in the cache are "checked out" by a given file. The buffer cache stores both a pointer to the file that requested the buffer and a pointer to a procedure to be called when the buffer must be reclaimed by the cache.

The buffer cache allows files to "pin" buffers so that they will not be flushed. Pinning is useful for holding inode data in the cache when files have other data in the buffer cache.

The buffer cache services a request for a buffer by requesting a memory block of the appropriate size from the memory manager. If the memory manager does not have enough free space, it tells the buffer cache to free enough space for the new buffer. The buffer cache flushes buffers from the cache until there is enough free space. The memory manager returns a memory block of the appropriate size, and the buffer cache checks the buffer out to the requesting file.

The buffer cache uses a variable page, LRU replacement scheme [105]. When a block of a particular size is requested from the cache, the buffer cache starts at the bottom of the LRU stack and scans upward marking buffers until it finds either a single buffer or an amalgamation of several buffers large enough to satisfy the request. It flushes the appropriate buffers and returns the space to the memory manager.

5.2.5 Directory-Name Cache

Lookup takes a directory (identified by its file handle) and a name and returns a file handle if the name is in the directory or an error otherwise. The directory-name cache reduces the number of accesses to directories. Entries in the cache are accessed by the directory-name pair and contain a file address. Directory-name pairs are looked up in the cache. If an entry is found, the file address is returned; if the entry is not in the cache, the cache looks in the directory for the entry. If the entry is in the directory, the entry is added to the cache and the appropriate file address is returned; otherwise an error is returned. The directory-name cache uses a simple demand-based insertion scheme and LRU-based flushing.

5.2.6 Memory and Disk Managers

The memory and disk managers both use buddy system allocation. The disk manager controls free disk space. It does not use a single free list, but rather a set of free lists. All blocks within a free list are considered equivalent, and requests may specify the free list from which the block should be allocated. The free lists are ordered by "performance," and requests may specify that, on allocation failure, the disk manager should scan the free lists in order of "increasing" or "decreasing" performance. The reason for this free list structure is described more fully in Section 7.5.

Chapter 6

Performance Evaluation

Measuring file-system performance is problematic. Each computer is used differently, so there is no single measure that predicts how a specific system will perform on each computer. Consequently, we used several existing benchmarks to evaluate the performance of iPcress.

The benchmarks are Andrew, Nhfsstone, and Bonnie, which are detailed in Section 2.4. Andrew measures the elapsed time to perform a variety of user-level tasks, such as copying a directory or compiling a program. Nhfsstone measures the average response time for a given mix of NFS operations, and Bonnie measures the disk throughput for creating, reading and writing a very large file.

Since iPcress is a single-threaded process, we compared its performance with that of a single ULTRIX NFS daemon. Doing so provides a fair comparison because both servers respond to one NFS request at a time. However, the ULTRIX NFS daemon uses prefetching for sequential scans and can flush dirty data from the buffer cache asynchronously, which iPcress cannot do.

6.1 Experimental Environment

The experiments were run on two nearly identical computers, one as the server and the other as the client, connected together on a local ethernet. We configured the iPcress and ULTRIX servers to be as similar as possible.

The server and the client were both DECstation 3100s with 24 Mbytes of memory. The server had three DEC RZ55 disks and one RZ56 disk, although only two of the RZ55 disks were used. The client had a single RZ55 disk. The ULTRIX NFS server was configured

Operation	Percentage
`lookup`	34
`read`	22
`write`	15
`getattr`	13
`readlink`	8
`readdir`	3
`create`	2
`setattr`	1
`remove`	1

Table 10: Nhfsstone Default Operation Mix

with 2.5 Mbytes of buffer memory (the UNIX buffer pool), which could not be modified, so we configured the iPcress server to use 2.5 Mbytes.

The client always mounted the file system (either iPcress or ULTRIX) in the same location with the same options, so that any overhead due to name lookup on the client would be uniform, and the servers would be accessed in the same fashion.

6.2 Nhfsstone

Nhfsstone measures both the average response time for each type of NFS operation and the overall average response time for all operations. Since there are many NFS operations, and since the average response time for the various operations is different, Nhfsstone's input is the distribution of NFS operations. The default distribution is shown in Table 10; operations that were not included in the input are omitted.

Figure 14 compares iPcress and ULTRIX on the Nhfsstone benchmark with the default mix and shows the overall average response time for each of the loads. The x axis measures load in calls per millisecond, where a call is any of the NFS operations. iPcress outperforms ULTRIX by roughly 6 ms per call.

6.3 Bonnie

Bonnie predicts file-system performance for multi-media applications, such as video images or digitized music. Typical multi-media data consists of very large files that are accessed

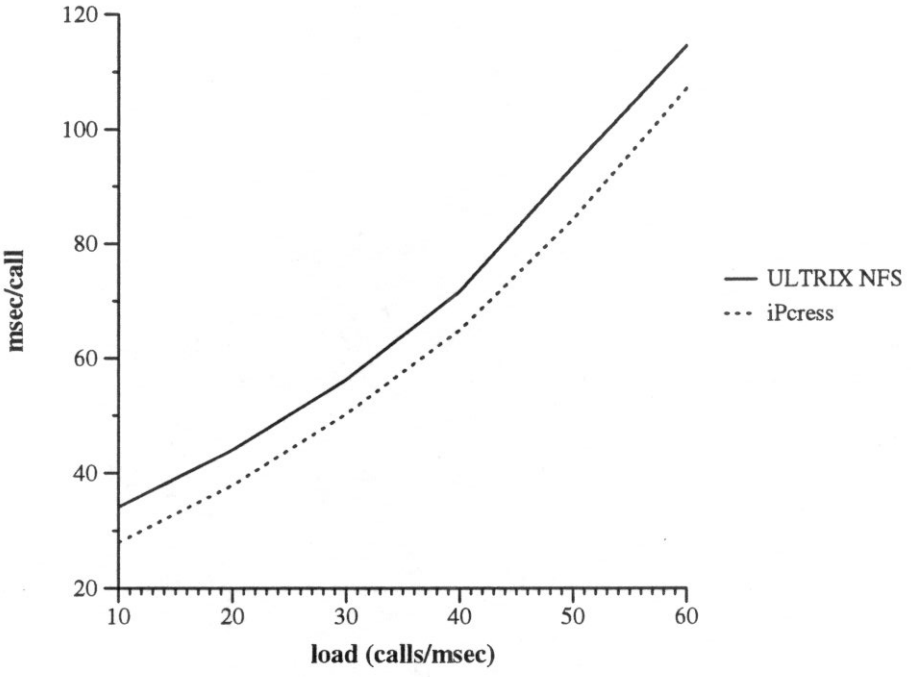


Figure 14: Nhfsstone Benchmark Results

sequentially. Bonnie mimics the behavior of such multi-media applications by creating a 100 Mbyte file and measuring the throughput to and from the file by reading and writing the file sequentially. During the reads and writes, Bonnie uses both character access and block access. Characters are transferred using the UNIX library functions `getchar` and `putchar`, and blocks are transferred using the `read` and `write` system calls.

Bonnie also measures system throughput for rewriting data, where the data is read, modified, and rewritten in place. A block is read into memory using `read`, some of its data is modified, and it is written back to the file using `write`.

Figure 15 compares iPcress and ULTRIX. iPcress outperforms ULTRIX on both character and block output, while ULTRIX outperforms iPcress on both character and block input. ULTRIX is better at sequential input because it has asynchronous read-ahead and can prefetch data from the disk. Both iPcress and ULTRIX perform poorly in rewrite mode because data is first read from disk through the server and then written back to disk through the server.

iPcress's performance is nearly identical for both reads and writes, while ULTRIX's reads are significantly faster than its writes. ULTRIX's poor write performance is probably

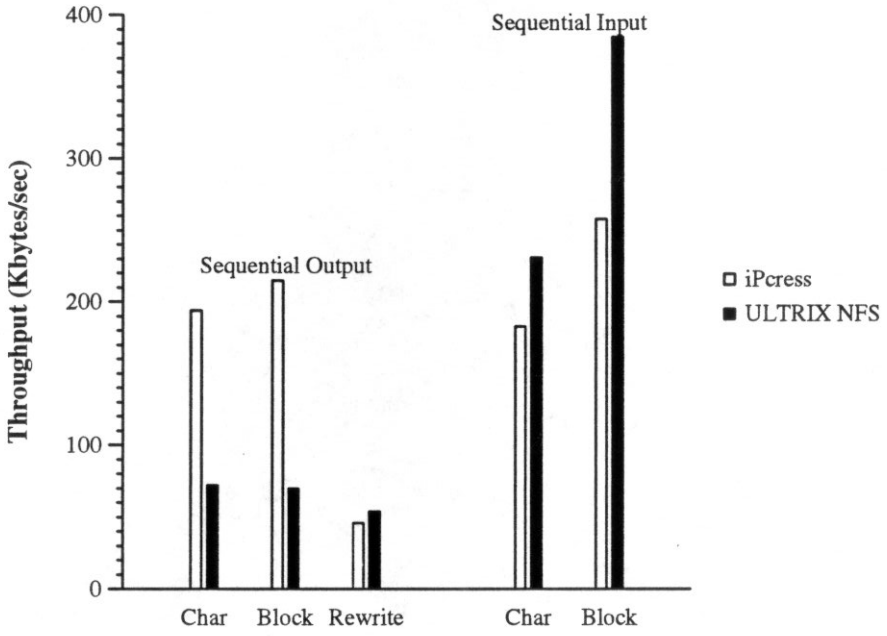


Figure 15: Bonnie Benchmark Results

related to its use of 4 Kbyte blocks; iPcress uses up to 32 Kbyte blocks.

6.4 Andrew

The Andrew benchmark has several phases, each of which represents a single "style" of access, and the benchmark measures only the time to complete each phase. The phases are described in more detail in Section 2.4 and in Table 3.

Figure 16 shows the performance of iPcress and ULTRIX on the Andrew benchmark. It shows the average time to complete each of the five phases in the benchmark. Phases 1, 2, and 5 create new files and data, phases 2 and 5 generate new data by reading old data, and phases 3 and 4 read only old data. iPcress is better on phases 1, 2, and 5, while ULTRIX is better on phases 3 and 4. Based on our observations using Bonnie, we expect these results for the Andrew Benchmark because of ULTRIX's prefetching using asynchronous read-ahead.

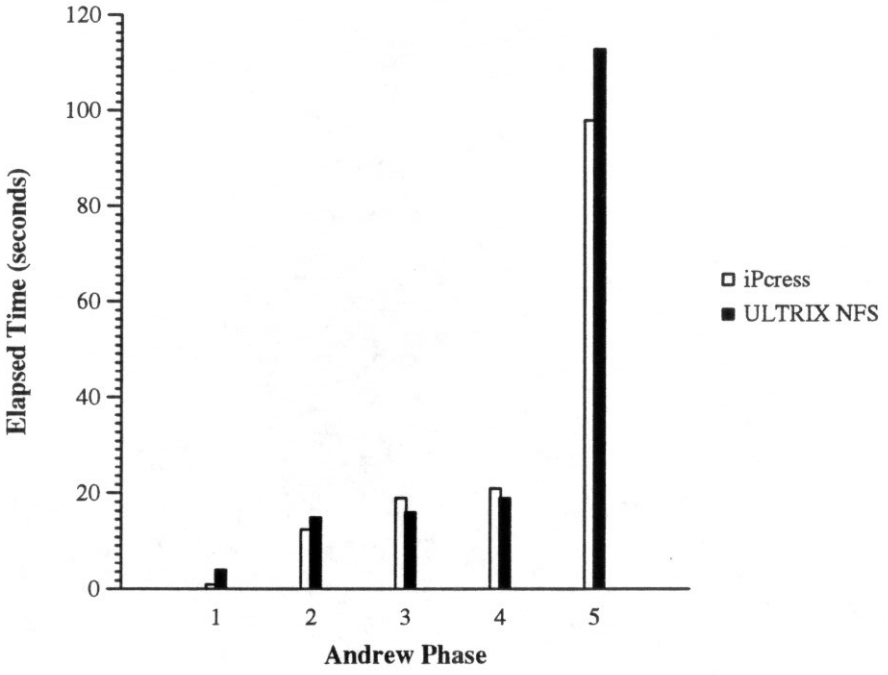


Figure 16: Andrew Benchmark Results

6.5 Performance Review

iPcress and ULTRIX perform about the same on Bonnie and Andrew, and iPcress does slightly better than ULTRIX on Nhfsstone. iPcress is a user process while the ULTRIX NFS implementation is part of the kernel. This difference adds overhead to each NFS operation in iPcress, since it must cross the kernel boundary at least three times per NFS request. It detects the request (using `select`), reads the request (using `read`), and replies to the request (using `write`). In addition, iPcress must cross the kernel boundary at least once, and often twice, for each disk operation.

Chapter 7

Data Clustering

Clustering the active data of a file system in the center of the disk might improve I/O performance by reducing the average seek delay. In the presence of large disk queues, clustering might reduce the average rotational delay. We present an implementation of a simple data clustering algorithm and show how it improves the performance of iPcress.

Chapter 3 shows that file-access patterns are strongly skewed and that file activity levels are relatively stable over time, making access patterns good predictors of future file activity. We simulate two techniques for reorganizing disk data and measure their sensitivity to imperfect predictions of future file activity. These simulations predict significant performance improvements over a full spectrum of use from light loads to heavy loads with large disk queues.

Our goal is to decide if, given the observed temperature distributions, the gains due to disk placement warrant the effort of keeping file statistics. We must also account for changing temperatures; temperatures drift slowly and it takes the system some time to reorganize the data. How sensitive are the expected improvements to this drift? Will the system have enough time to reorganize the data?

7.1 Access Model

We developed a model for disk accesses based on the file-access pattern results in Chapter 3. Varying this simple model helps gauge the effects of various factors on the performance benefits of data clustering.

Since data clustering reduces the seek distance, and seek time is a function of head

movement over cylinders, we model cylinder access probabilities, which are generated from the cumulative space versus cumulative I/O curve in Figure 17. To simplify the analysis, we assume that accesses are independent and have uniform size.

We model two situations: files are scattered randomly over the disk, and files are placed perfectly according to their temperature. In the first case, the cylinder access probabilities are uniform, and in the second case the files are packed perfectly, so that each byte is in the right cylinder.

Given a disk with cylinders numbered $1, 2, 3, ..., N$, the probability of accessing each cylinder is

$$Prob[\text{next access is to cylinder } x] = f(x) \quad x \in \{1, 2, \ldots, N\}$$

When data is placed randomly on the disk, the probability of accessing any given cylinder is

$$f_r(x) = \frac{1}{N} \tag{1}$$

When data is placed "perfectly" according to temperature, the hottest data is placed in cylinder $N/2$, the next hottest in cylinders $(N/2) - 1$ and $(N/2) + 1$, and so on.

We assume that temperatures are strongly skewed and use the cumulative distribution, $G(s)$, given by Reference [119] for generating the access distribution for perfect placement; Figure 17 reproduces this distribution.[1]

Using the cumulative distribution $G(s)$ described above, f_p for an N-cylinder disk is

$$f_p(x) = \begin{cases} G(\frac{1}{N}) & \text{if } x = \lfloor \frac{N}{2} \rfloor \\ \dfrac{G(\frac{2|\frac{N}{2} - x| + 1}{N}) - G(\frac{2|\frac{N}{2} - x| - 1}{N})}{2} & \text{if } x \neq \lfloor \frac{N}{2} \rfloor,\ 1 \leq x \leq N \end{cases} \tag{2}$$

The hottest $\frac{1}{N}^{th}$ of the data is in the center cylinder ($x = \lfloor \frac{N}{2} \rfloor$). The next hottest $\frac{2}{N}^{th}$ of the data is spread evenly between the two cylinders on either side of the center cylinder ($x = \lfloor \frac{N}{2} \rfloor \pm 1$). Since $G(s)$ is cumulative, the probability of accessing data in the second hottest cylinders is $G(3/N) - G(1/N)$, and the probability of accessing one of these cylinders (say $x = \lfloor \frac{N}{2} \rfloor + 1$) is that quantity over 2.

[1] Reference [119] used the same data as our study [110].

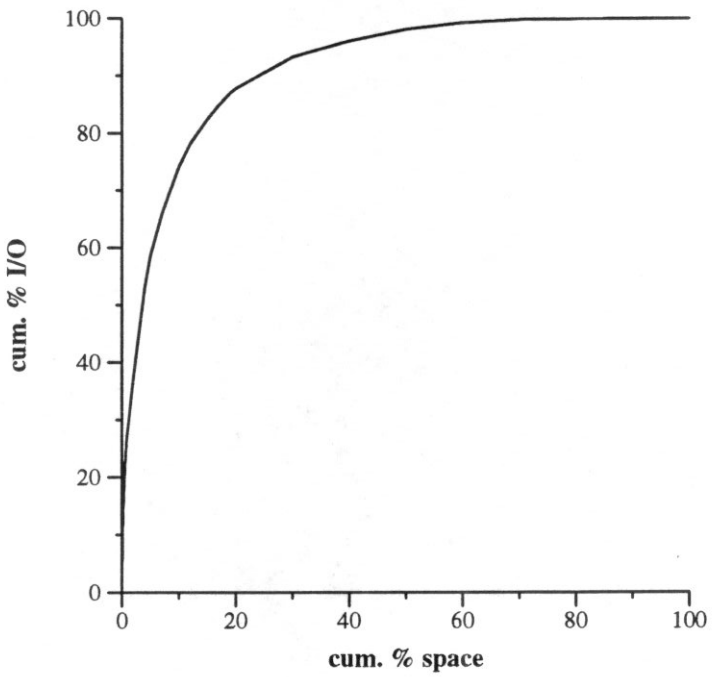


Figure 17: Cumulative Space vs. Cumulative I/O

Another distribution is motivated by the observation (from Chapter 3) that many files are simply untouched, and there is empty disk space. Some fraction of the disk is rarely used, and that data can be moved to its edges. Performance improves because the disk arm will move only among the cylinders used.

Suppose that only $M = \lfloor kN \rfloor$ cylinders in the disk are used during a given test period, where $0 < k \leq 1$. Thus, the range of active cylinders is $a = \lceil \frac{N}{2} \rceil - \lfloor \frac{M-1}{2} \rfloor$ to $b = a + M - 1$. Within this range, the probability of access is as if there were only M cylinders, and cylinder a was actually the first cylinder. Taking f_p^M as f_p from Equation 2 with M substituted for N, the access probability with such "concentrated" data is

$$f_c(x;\ k) \ = \ \begin{cases} f_p^M(x) & \text{for } a \leq x \leq b \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

The file system attempts to place files by their temperature. However, when the temperature of a file changes, it takes time to recognize this change and additional time to actually move the file. To model the effect of drifting temperatures, let us say that a file that has not drifted will be accessed with probability $(1 - \delta)$; with probability δ the accessed file has

drifted and is hence located in a random spot on the disk. This effect can be modeled with the distribution $f_d(x\,;\,\delta) = (1-\delta)\,f_p(x) + \delta\,f_r(x)$. This model is probably pessimistic, and drifting accesses will probably not be distributed uniformly over the disk.

We have assumed that if a file has a temperature corresponding to cylinder x, it is placed in that cylinder. In practice, such perfection may be hard to achieve because that cylinder may be full or it may be too expensive to make room. A more effective approach divides the cylinders into buckets and places files in the bucket that corresponds to these temperatures. To model this approach, $b(x)$ maps a cylinder number to one of the buckets. For example, $b(1)$ is the first bucket, i.e., the one for the hottest cylinders. Depending on the size of the bucket, cylinders 2 and 3 may be in this bucket ($b(2) = b(3) = 1$), and so on. Within a bucket, data is placed at random. Thus,

$$f_b(x\,;\,b) = \frac{\sum_{\{i \mid b(i)=b(x)\}} f_p(i)}{|\{i \mid b(i) = b(x)\}|} \tag{4}$$

Finally, suppose the system focuses on only the hottest fraction k of the data. If the data is hot enough, it will be placed in the correct cylinder by temperature; otherwise it will be placed randomly in the remaining $(1-k)$ fraction of the disk. There will be a total of $M = \lfloor kN \rfloor$ cylinders in the reorganized area. The cylinders in this case will be $a = \lceil \frac{N}{2} \rceil - \lfloor \frac{M-1}{2} \rfloor$ through $b = a + M - 1$. This case models a "lazy" system that partially organizes data. The access distribution is

$$f_l(x\,;\,k) = \begin{cases} f_p(x) & \text{for } x \in [a,\ldots,b] \\ 1 - \sum_{i \in [a,\ldots,b]} f_p(i) & \\ \hline N - \lfloor kN \rfloor & \text{otherwise} \end{cases} \tag{5}$$

These models do not take into account that files tend to be accessed sequentially and that many files are bigger than a single, fixed block size. However, we could assess the effect of sequential access using a single parameter that represents the probability of any given access accessing the next block on the disk. The results below may be interpreted as having zero probability of intentional sequential access.

7.2 Simulator

The simulator takes as input a function describing the cumulative disk space versus cumulative I/O. In our simulations, we use the distribution shown in Figure 17 and a uniform distribution. We model drifting temperatures and concentrated data by applying f_d and f_c to the perfect distribution and using the resulting distribution and the uniform distribution as input to the simulator.

Using the input described above and a description of the disk, the simulator generates a table of cylinder access probabilities. The method used to generate this table depends upon the type of reorganization scheme being modeled. The simulator can model four reorganization schemes: none, perfect, partial, and buckets, the distributions for which are given by f_r, f_p, f_l, and f_b, respectively.

The simulator is based on the one used in Reference [102] with the extensions described above regarding non-uniform cylinder access probabilities. A single run of the simulator consists of initialization, filling the disk queue, steady state servicing of the disk queue, and flushing the disk queue. During each run, the simulator "services" a fixed number, n, of disk requests, and the disk queue is maintained at a fixed size, q, by adding new requests as each request is serviced until n requests have been generated. Then, the simulator continues to service requests until the queue is empty.

For each combination of parameter values, we ran the simulator several times with different seeds for the random-number generator and analyzed the variability of the results. The run length was large enough (10,000 requests) to reduce the variability to less than a few percent. The queue length is an input to the simulator. "Edge effects" like filling and flushing the queue are negligible because the number of requests (10,000) is large relative to the maximum queue length (200).

The simulations use a variety of disk scheduling algorithms: FCFS, CSCAN, SSF, STF, WSTF, and FGSTF10. FCFS is the simple first-come-first-served algorithm. CSCAN is a cylindrical scan algorithm where the head scans the disk servicing requests as it goes and jumps back to the start of the disk at the end of a scan. SSF is the shortest-seek-first algorithm, where the disk always services the request with the smallest seek time, regardless of rotational position. STF is the shortest-time-first algorithm and the disk always services the request with the shortest service time, including both seek and rotational delays. WSTF is similar to STF except that the expected service time calculation includes a negative term

Attribute	Value
cylinders	1224
tracks/cylinder	15
sectors/track	36
bytes/sector	512
RPM	3600

Table 11: RZ55 Disk Geometry

for time spent in the queue, which reduces the maximum delay experienced by a request. FGSTF10 is a grouped-shortest-time-first algorithm; the disk is divided into 10-cylinder groups and requests within a group are serviced using an STF algorithm, but decisions regarding which group is serviced next are based on a SCAN algorithm, which scans the disk from one end to the other and then scans in the other direction.

The simulator provides summary statistics regarding the average time required to service a request, the average turn-around time for a request, and the maximum turnaround time for any request.

The simulator has descriptions of several disks encoded and we have results for several of them. Since many of the results were qualitatively similar, we show results for only the DEC RZ55 disk. The simulator's geometry information and the seek cost function are used to compute the seek and rotational delays. Table 11 lists the RZ55's geometry. The seek cost function is

$$C(d) = \begin{cases} 0 & \text{if } d = 0 \\ 1.49 + 1.18\sqrt{d} - 0.015d & \text{otherwise} \end{cases} \quad (6)$$

$C(d)$ is the seek cost in milliseconds, where d is the seek distance in tracks. This equation was derived from experiments that measured the average access time for 1,000 accesses at each of three seek distances (1, 5, and 800 cylinders) and the average access time for 1,000 accesses with no seek distance. Subtracting the second value from each of the first values gives the average seek time for the three seek distances, which led to the coefficients in Equation 6.

7.3 Simulation Results

As shown below, the relative performance of the various disk scheduling algorithms is the same with and without reorganization. If algorithm A is better than B but worse than C before reorganization, these relations hold after reorganization. Unless otherwise stated, each simulation is for a blocksize of 4,096 bytes (8 sectors on an RZ55).

The performance measure is *disk efficiency*, which is the ratio of the time the disk spends transferring data over the total time the disk is active. 100% efficiency means that, during each I/O, the disk spent all of its time transferring data.

Figure 18 plots disk efficiency versus queue length assuming that the disk has not been reorganized and that accesses are uniformly distributed. We compare the performance of reorganized data against this case. Algorithms WSTF and STF perform best, followed by FGSTF10, CSCAN, SSF, and FCFS. Also, the efficiency for all algorithms except FCFS increases quickly for queue lengths less than 20.

Queue lengths of over 200 have been observed in UNIX systems [70]. We expect large disk queues because of delayed writes and prefetching. Queues will likely become prioritized, with high priority for delay-sensitive requests, such as demand fetches, and lower priority for delayed writes and prefetching. Large caches accumulate dirty data and flushing them will lengthen queues as will optimistic prefetching to minimize disk delays. These trends will increase the average queue lengths over the next few years.

Figure 19 shows the disk efficiency versus queue length for the same scheduling algorithms as Figure 18, except that the data has been reorganized perfectly. We assume that all data is accessed (all cylinders have a non-zero access probability), and that the accesses are distributed according to the curve in Figure 17. The curves in Figure 19 have the same shape as the corresponding curves in Figure 18 except that they have all been moved upward. For example, Figure 18 shows FCFS has an efficiency of about 0.11 but it has an efficiency of about 0.17 in Figure 19.

Figure 20 displays the maximum response times in milliseconds for the scheduling algorithms versus the queue length. The maximum response time is the maximum time required to service a request including both queueing and service delays. WSTF and STF have long response times because requests near the edges of the disk starve while the head services all of the requests near the center.

In most existing systems, the maximum response time is important, since many jobs

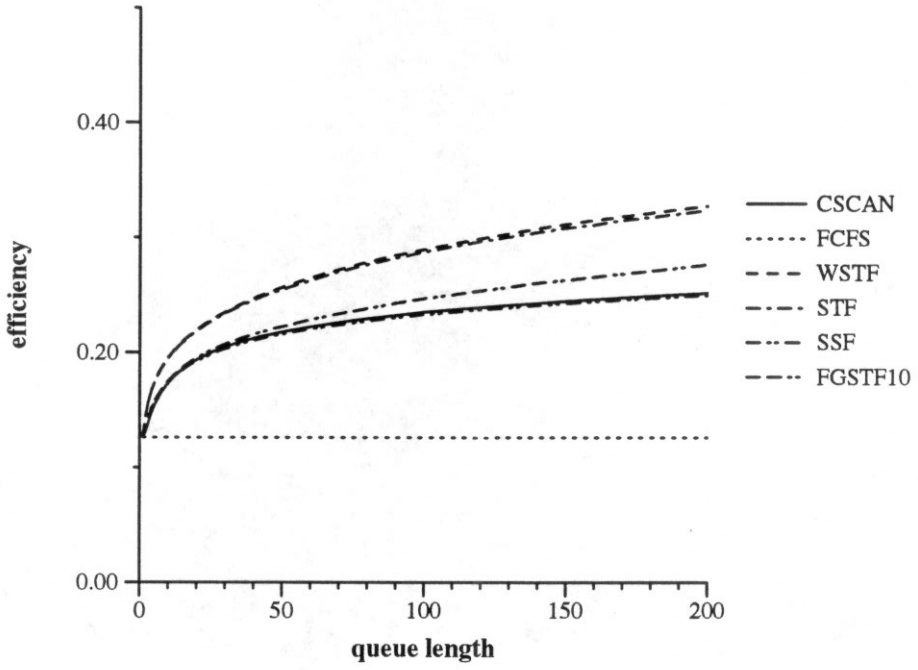


Figure 18: Efficiency for Uniform Distribution

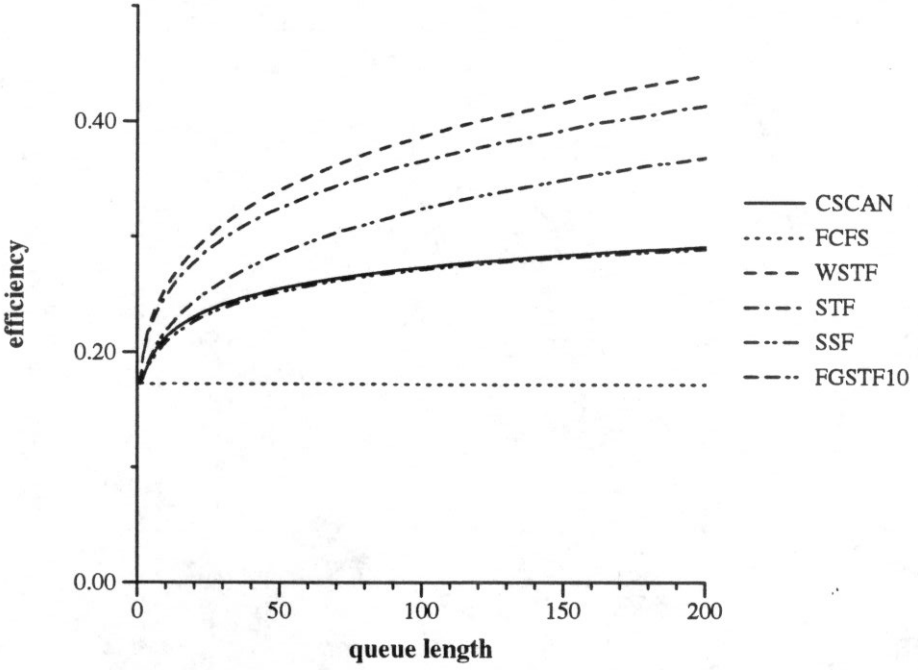


Figure 19: Efficiency for Perfect Distribution

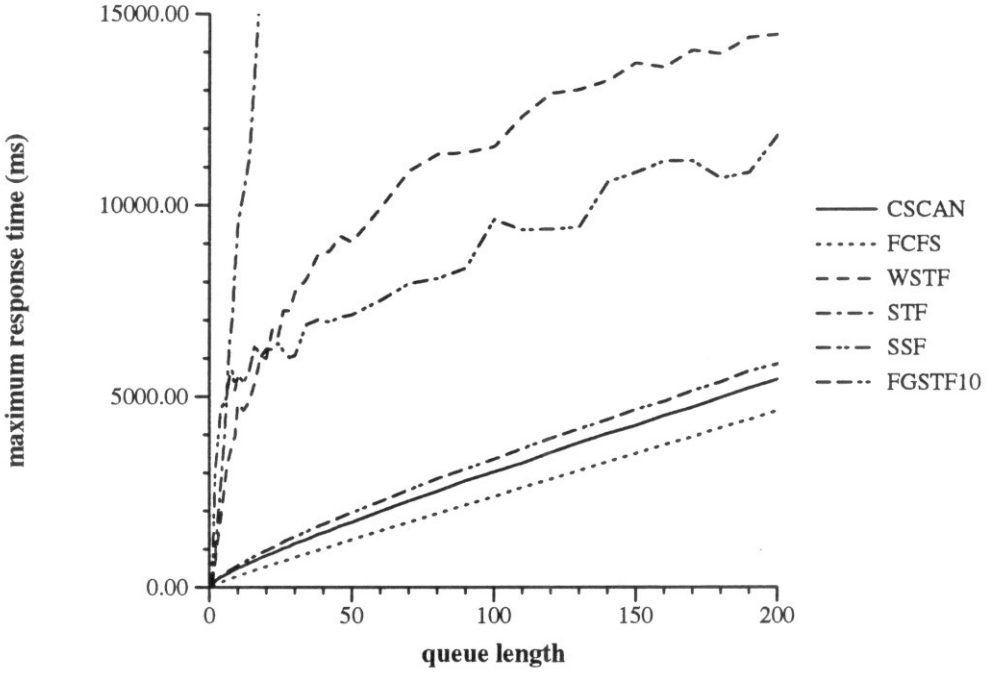


Figure 20: Maximum Response Time for Perfect Distribution

wait for I/O to complete before continuing. In future systems, this metric may become less important as non-volatile memory allows systems to delay writes indefinitely and focus on system efficiency rather than maximum response time. In addition, systems may incorporate priority queueing to ensure that response-time sensitive requests get serviced first.

Figures 18 and 19 show that WSTF and STF have the highest efficiencies, but also have large maximum response times. FGSTF10 is nearly as good as the best algorithms and has a maximum response time close to that of CSCAN. The reason for this behavior is that FGSTF10 is a SCAN algorithm at the group level, so requests near the edge of the disk do not starve, and it is also an STF algorithm so that requests near the center are serviced quickly.

The speedup from reorganizing the data is

$$\frac{\text{Efficiency [clustering]} - \text{Efficiency [base case]}}{\text{Efficiency [base case]}}$$

Figure 21 shows this speedup. The best algorithms, WSTF, STF, and FGSTF10, get 20–27% speedup from reorganization.

The remaining simulation results evaluate the impact of drifting temperatures, dormant

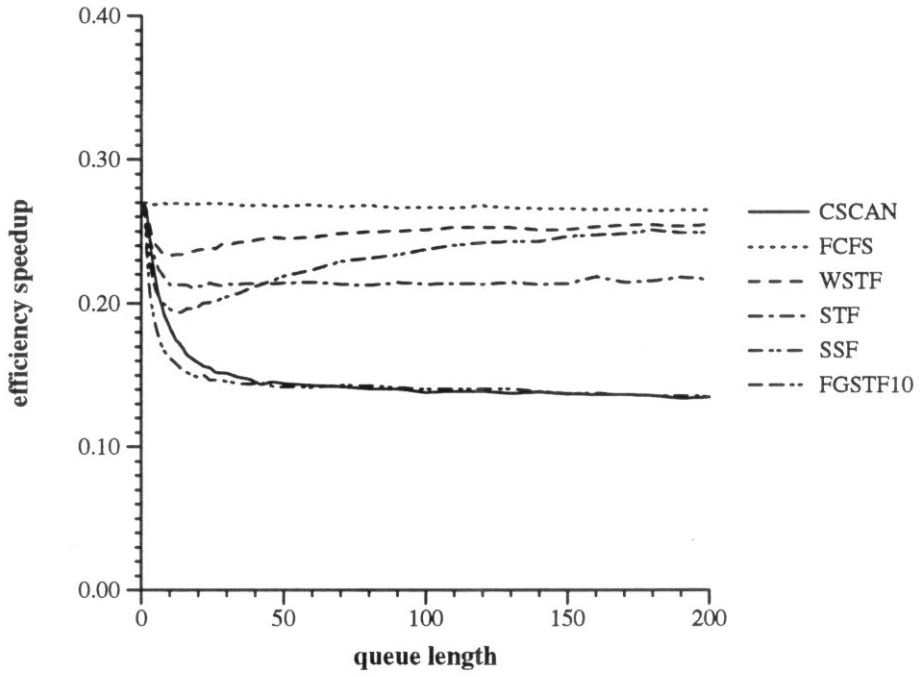


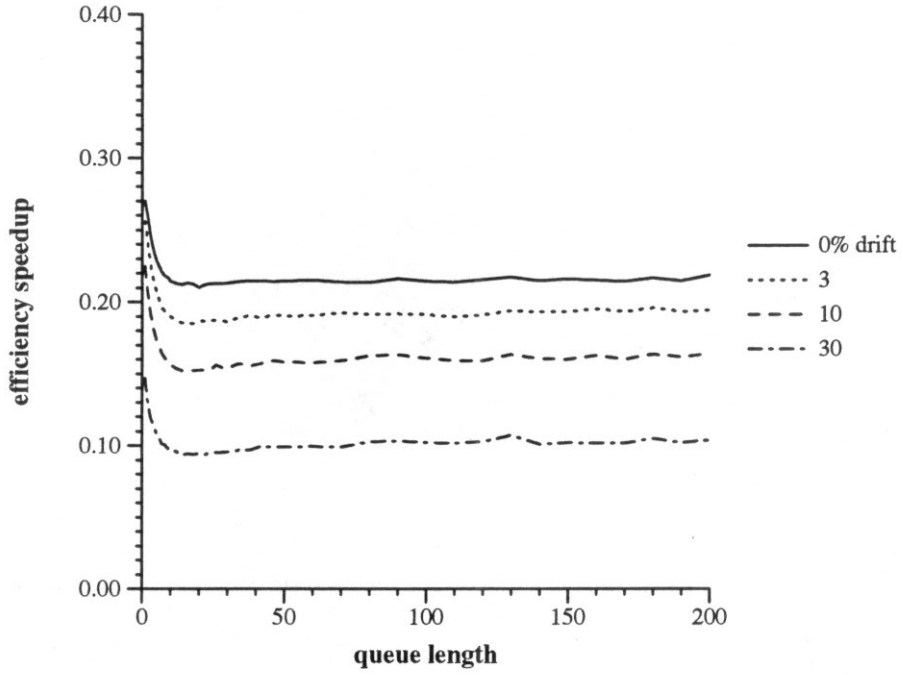
Figure 21: Efficiency Speedup for Perfect Distribution

files, and sequential I/O. The relative performance of the various scheduling algorithms is similar, so only the results for STF are shown in the following figures.

Figures 22 and 23 display the impact of drifting temperatures on reorganization performance. We assume that some fraction of the I/O is distributed uniformly over the disk, while the remaining I/O is distributed perfectly according to the distribution from Figure 17; the cylinder access probabilities are from f_d. As the drift fraction reaches 10%, roughly 30% of the speedup from reorganization is lost; when the drift fraction reaches 30%, roughly 50% of the speedup is lost.

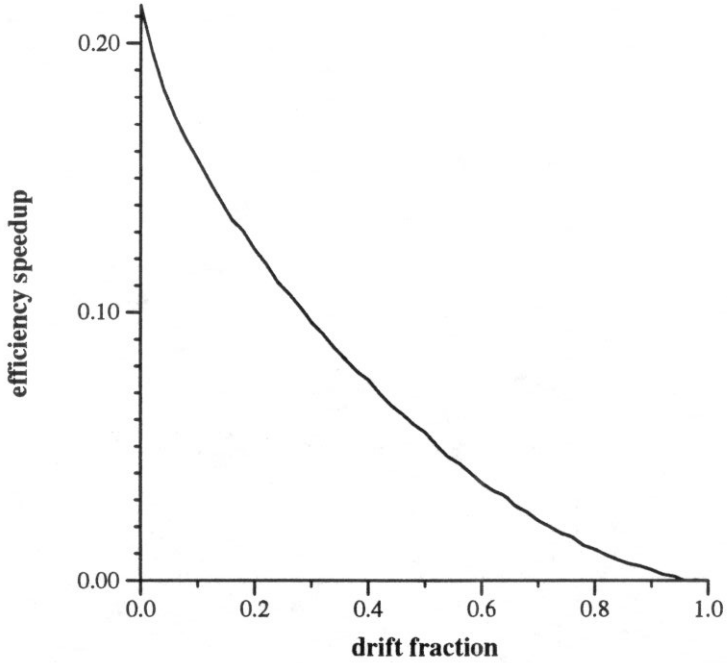
Figure 23 demonstrates the efficiency speedup versus drift for a 10-element queue. The speedup drops nearly linearly with respect to drift. This drop is expected because the drift represents uniformly distributed accesses, which cannot be improved by clustering. As the fraction of these random accesses increases, the fraction of improved access drops correspondingly.

Figures 24 and 25 show how dormant files affect system performance. Most files (roughly 70–80%) are dormant, so we simulated performance at 50%, 80%, and 90% dormant files using Equation 3 for the cylinder access probabilities. The speedup with 80% of the files



STF scheduler

Figure 22: Efficiency Speedup for Drift Distribution



STF scheduler, Queue=10

Figure 23: Efficiency Speedup versus Drift Fraction

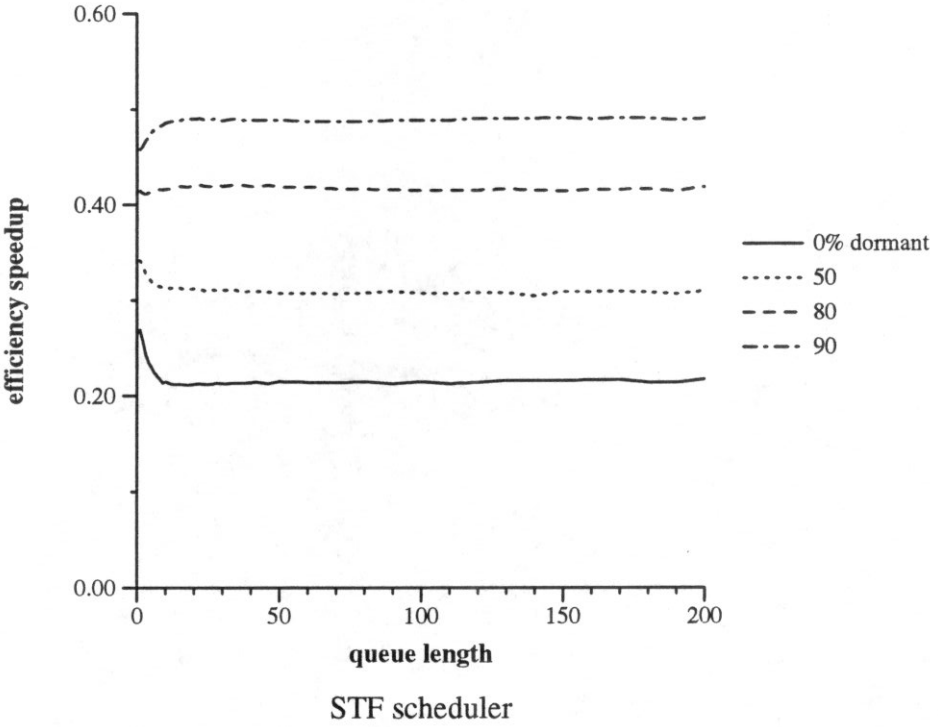


Figure 24: Efficiency Speedup for Dormant Distribution

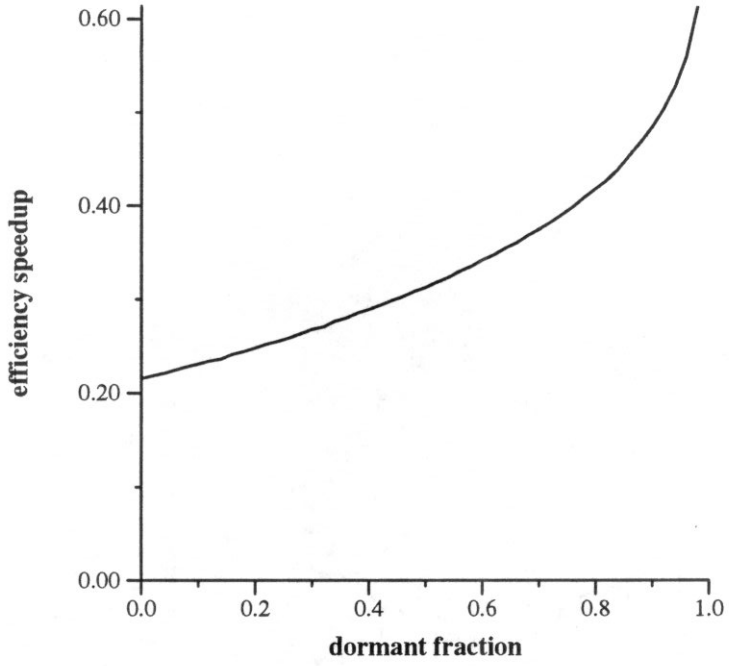
dormant is roughly double that of the perfect reorganization without any dormant files (Fig. 21).

Figure 25 illustrates the efficiency speedup versus the dormant fraction for a 10-element queue. Speedup improves slowly until roughly 80% of the data is dormant, and then improves quickly. This behavior is probably due in part to the non-linear nature of the seek cost function for the RZ55 (Equation 6).

Figures 26 and 27 depict the impact of sequential I/O on efficiency. Sequential I/O to large files may be represented by a *sequentiality* fraction, which represents the probability that the next disk access will be to the block following the current block. This addition removes the assumption of independent disk accesses, but we still assume that all non-sequential accesses are independent.

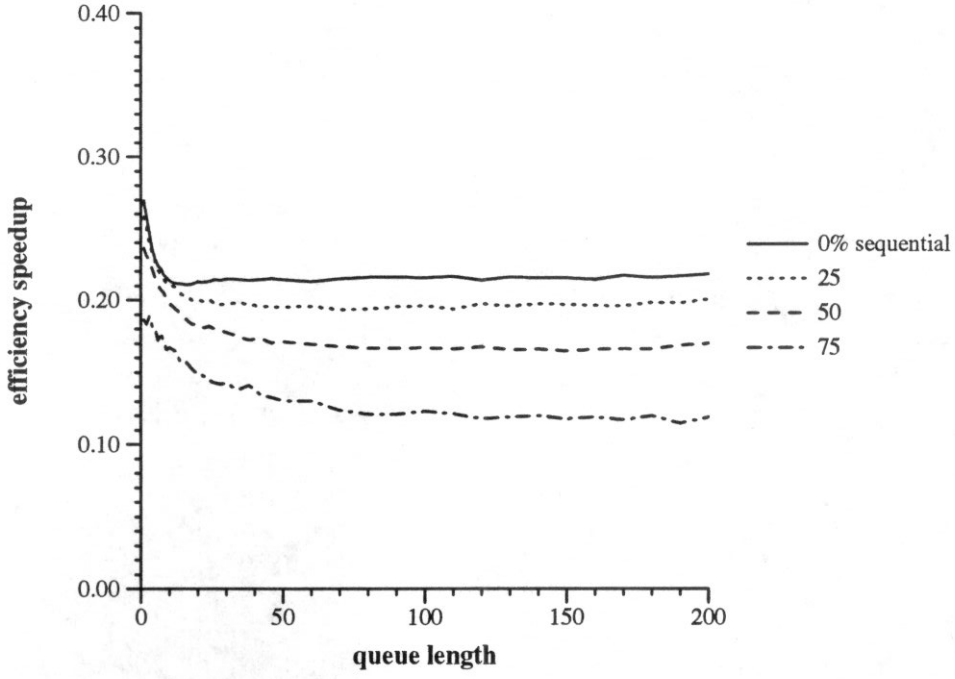
Data clustering improves seek delay, and sequential I/O increases the transfer delay without increasing either seek or rotational delays. Consequently, speedups due to reorganization should diminish as sequentiality increases.

As Figure 27 reveals, speedups are not very sensitive to sequentiality; 60% sequentiality retains roughly 90% of the speedup.



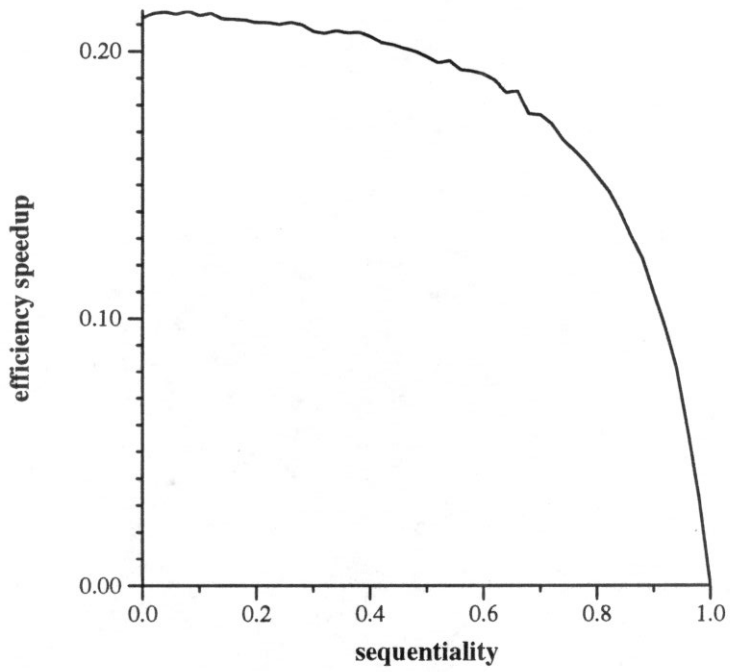
STF scheduler, Queue=10

Figure 25: Efficiency Speedup versus Dormant Fraction



STF scheduler

Figure 26: Efficiency Speedup versus Sequentiality



STF scheduler, Queue=10

Figure 27: Efficiency Speedup versus Sequentialities

7.4 Reorganization Strategies

Our objective for implementing a clustering strategy is to provide the maximum performance with the minimal reorganization. We expect disks to be reorganized (at least) daily and file temperatures to drift only slightly between reorganizations. We evaluated the three reorganization algorithms perfect (f_p), partial (f_l), and buckets (f_b).

Figure 28 shows the efficiency for these algorithms with a queue of length 1. The leftmost point is the base or uniform case. The bar just to the right of the uniform case is the perfect case. Just to the right of the perfect case is a set of bars for various numbers of buckets. The rightmost set of bars shows the efficiency for partial reorganization with various percentages of the disk reorganized perfectly. Note that both partial and buckets perform about as well as perfect reorganization with a reasonable reorganization fraction or number of buckets, respectively.

The time used to reorganize the disk is an important criterion. Since file temperatures drift slowly, bucket reorganization will likely require the least work on a nightly basis. Bucket reorganization would not move most files because they should already be in the

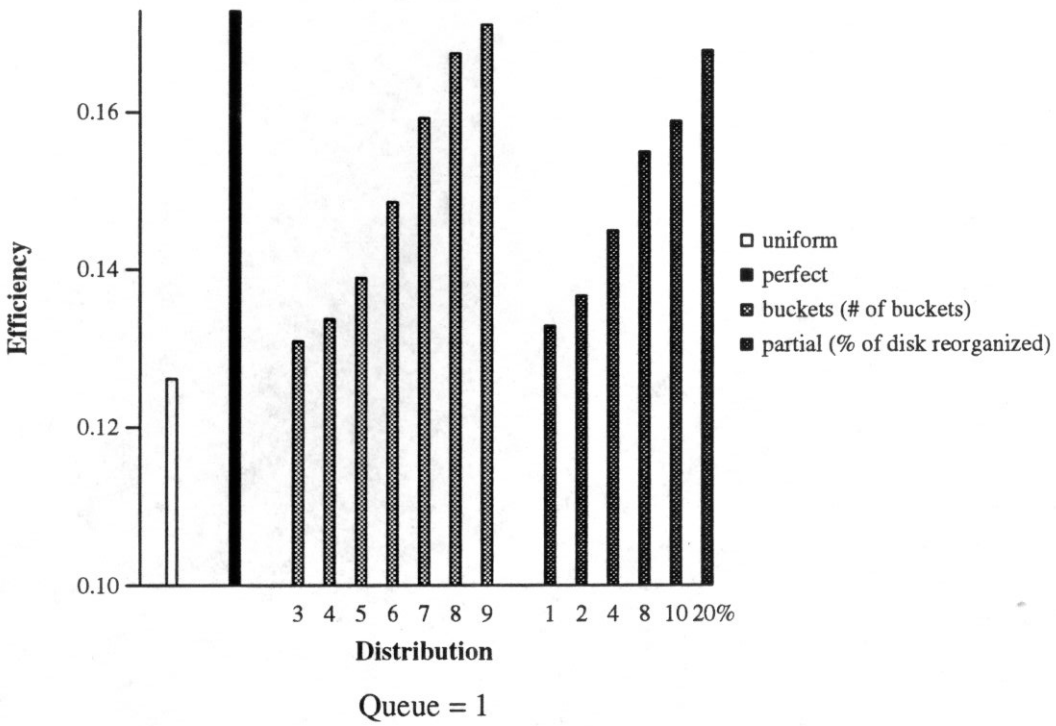


Figure 28: Simulated Efficiency for various Reorganizations

correct bucket; the other reorganizations would most likely move all files in the active portion of the disk.

7.5 Data Clustering Implementation

iPcress has a simple "proof of concept" batch-oriented data clustering algorithm. It uses file activity statistics to rank files by their relative activity. The system moves all files with no accesses (dormant files) as close to the edges of the disk as possible and moves active files as close as possible to the center of the disk based on the ranking.

iPcress measures file temperature by approximating the time the disk spent servicing requests for each file. We can not use the number of I/Os over the number of blocks allocated to the file because iPcress uses variable size blocks. If two files have identical access densities and size but have different buffer sizes, the file with the smaller buffer size will usually require more time. Consequently, we weight the file temperature by the number of disk accesses and the number of sectors transferred on behalf of the file. The temperature measure is

$$T = \frac{CN_{\text{disk accesses}} + N_{\text{sectors transferred}}}{N_{\text{sectors allocated}}}$$

iPcress measures file activity with a five-day moving average of T. C is a constant used to weight $N_{\text{disk accesses}}$, the number of disk accesses performed on behalf of the file. $N_{\text{sectors transferred}}$ is the number of 512-byte sectors transferred, and $N_{\text{sectors allocated}}$ is the number of sectors allocated to the file exclusive of the inode. C is 32 and was chosen so that one 512-byte access had roughly half the weight of one 16-Kbyte access, since the average time spent for a 512-byte access is roughly half that of a 16-Kbyte access.

Disk space is divided into buckets of unequal size. The buckets are symmetrical about the center of the disk and grow exponentially in size as one moves away from the center. For example, the center bucket consists of a single cylinder, and the next bucket consists of two cylinders, one on each side of the center bucket, and so on. Blocks within a given bucket are considered equivalent. The file system tries to place files in an appropriate bucket.

The reasoning behind the exponential bucket size is that there are few hot files and they must be placed in the correct cylinder. However, as we move away from the center of the disk files are not accessed as frequently, so they merely need to be within a few cylinders of their optimal location. At the edge of the disk are the dormant files, which may use 50–80% of the space, and these files merely need to be away from the center.

We have not described *how* (or when) the file temperature ranking is created, how free space is divided among the various buckets, or what happens when there is not enough free space. Finally, we have not described what happens when a single file is too large to fit in its assigned bucket.

The file ranking is decided at the outset of the reorganization and is fixed for the duration of the reorganization. It is created by scanning the inodes and creating a sorted list of all files with disk space allocated and with non-zero file temperature.

iPcress has no fixed policy for free-space placement. It leaves at least 10% of each bucket free. Extra free space appears in the larger buckets, since the smaller buckets are filled to their limit (90% full).

When there is plenty of free disk space, a simple reorganization strategy copies all data as close to the edge of the disk as possible, which leaves the center of the disk empty. Then, in descending order of file temperature, files are moved as close to the center of the disk as possible.

As a disk fills up and the free space shrinks, disk reorganization becomes more complicated. It becomes impossible to empty entire buckets. In addition, iPcress uses variable size blocks, so a bucket with enough total free space may not have a free block large enough for a specific block. Consequently, when a file must be moved into a bucket that does not have enough free space, another file (with lower temperature) may have to be moved out of the bucket.

It is best to get files as close to their assigned buckets as possible. Files can move towards the center or towards the edge. In the first case, a file should land as close to the center as possible by attempting to place it in successive buckets moving outward, and it should certainly be placed no further from the center than its original location. In the second case, a file should land as close to the edge as possible by attempting to place the file in successive buckets moving inwards. Large files may not fit in their assigned bucket, and files can be larger than the cumulative size of several buckets. In these cases, the file is simply placed in its assigned bucket and as many successive buckets as necessary.

The disk reorganization algorithm has four stages. The first stage initializes the temperature ranking and moves dormant files to the edge of the disk. The temperature ranking is a sorted list of file temperatures and internal addresses for all active files. The second stage computes the optimal bucket for each file using a greedy algorithm: beginning with the hottest files and the center-most bucket, it assigns as many files to the bucket as possible, and so on. No files are moved in this stage.

The third stage runs through the list of files, starting with the coldest files, and attempts to place each file in the correct bucket. If a file is too close to the center, but there is no room in the appropriate bucket, it is placed as close to the assigned bucket as possible, but no closer to the center than its assigned bucket. This stage moves files that are too close to the center towards the edge, freeing needed space near the center.

The fourth stage tries to place each file in its assigned bucket. Due to free space fragmentation, it may not always be possible to place a file in its assigned bucket, so this stage ensures that only the coldest files in each bucket are moved out of the bucket into the next bucket when necessary. The fourth stage runs over the list of files in order of decreasing temperature attempting to place each file in its assigned bucket. If the current file is the coldest file in a bucket, a single attempt to place the file in the correct location is made. When a file cannot be placed in its assigned bucket, the coldest file in the bucket is reassigned to the next bucket outward from the center and the fourth stage is repeated for the

original file.

This algorithm is slow, but it works with variable block sizes. In a production environment, more effort would be needed to improve the running time, particularly when there is little free disk space.

7.6 Benchmark

We evaluated the data clustering facility with a new benchmark, FSBench, which emulates the strongly skewed nature of file-access patterns. The benchmark consists of a *generator*, which creates a table of file-access probabilities, and a *driver*, which uses this table to read data with skewed probability.

The generator takes as input a directory name (the root directory of a sample file system), a parameter that describes what fraction of the system is dormant, the amount of data to be read during an experimental run, and a distribution that describes the access pattern to the active data. The input distribution, taken from experimental observations, specifies the cumulative distribution for I/O versus space that the generator should match. The generator produces a table of file names and a number of times each is accessed.

Based on the dormant fraction, the generator randomly assigns files to be dormant or active and guarantees that the cumulative size of the dormant files is as close as possible to the cumulative size of the file system times the dormant fraction. The generator also guarantees that dormant files are not accessed by the driver by omitting them from its output.

For the active files, the frequency of access is generated using the cumulative input distribution shown in Figure 17. It is guaranteed that each active file will be accessed at least once by the driver. The active files are assigned a random ranking. Using this ranking, the size of each file, the cumulative size of all active files, and the total number of bytes to be accessed during each run, the generator can assign an access count to each file based on the cumulative input distribution. It is assumed that each access transfers the entire file.

Unless the number of bytes accessed during each experimental run is several times the size of the entire file system, the distribution created by the generator will not be as skewed as the input distribution, because each active file must be accessed at least once during each run. Our experiments accessed ten times as many bytes as were contained in the file system.

The driver reads the table of file names and the number of accesses, randomly chooses a file from the table, reads it, and decrements its access count. The probability of choosing a particular file is the number of accesses remaining for that file over the total number of remaining accesses. Thus, at the end of a run, all files have been accessed the specified number of times. The driver reports the time necessary to process all of the accesses.

7.7 Performance

In order to measure performance improvements, we modified iPcress to measure the cumulative time spent doing disk I/O and the number of disk operations. The experiment measures the speedup in disk efficiency and disk response time due to reorganizing the disk. It uses a sample file system for the experimental file system accessed by the benchmark. The sample file system contains system and kernel sources, objects, and documentation, roughly 210 Mbytes in nearly 16,000 files.

The benchmark and iPcress reported a variety of performance statistics during the experiments. The benchmark driver reported the time required to access all of the data (as determined by the table of file accesses). For each disk block size, iPcress reported the number of disk accesses and the cumulative time required for those disk accesses. In addition, iPcress reported the cumulative time spent on all disk accesses, the total number of bytes transferred, the total time spent servicing each NFS operation, and the total time spent reorganizing the disk.

Each run of the experiment measures the performance improvement after a single reorganization. First, the system copies the sample file system into a new file system, then the generator creates the table of file accesses to be used by the driver. At this point, the driver accesses the file system the first time. When the driver finishes, both the driver and the file system report the first set of performance statistics, and iPcress reorganizes itself. Finally, the driver accesses the reorganized file system again and the final performance statistics are reported.

The experiment used the iPcress NFS server and the NFS client. Both machines are DECstation 3100's with local disk on the same thick-wire ethernet subnet. The subnet is one of Princeton's primary departmental subnets with over 60 X terminals and machines. During some of the experiments, packet collision rates exceeded 15%, so we discarded the performance results from the client for these runs.

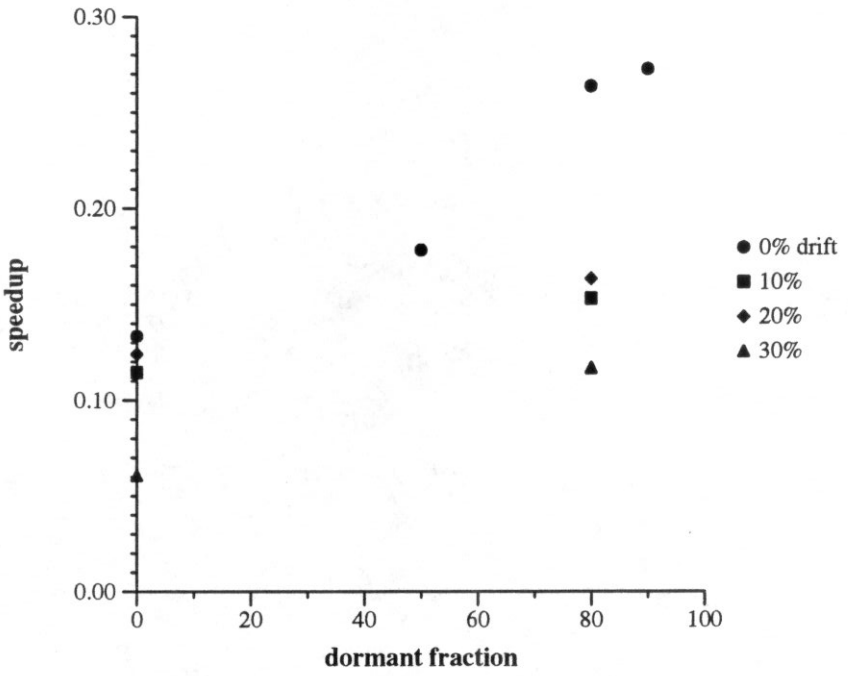
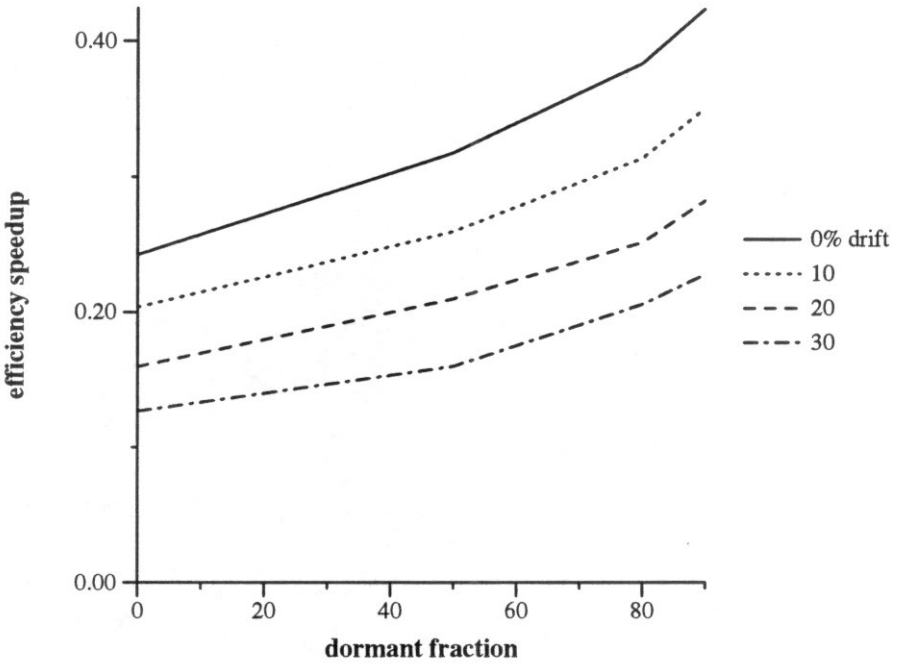


Figure 29: Efficiency Speedup

The disk is an RZ55 (see Table 11). iPcress accesses the disk via the UNIX raw device, which has roughly 310 Mbytes available. Under iPcress, the sample file system had 275 Mbytes of file system data and 35 Mbytes of free space.

The mapping of the UNIX device-driver blocks to physical disk blocks, including the bad-block mapping, is invisible to iPcress. Consequently, blocks that are adjacent at the UNIX device-driver level may be widely separated on the disk. Each disk has its own unique pattern of bad blocks, so the effect of bad-block remapping on data clustering will be different for each disk. If most of the bad blocks are near the edges of the disk, this effect can be ignored. If there are bad blocks near the center of the disk, then the remapping of disk blocks may have a noticeable, negative impact.

Figure 29 shows the speedup in efficiency, where efficiency is the transfer rate, which is measured by recording the size and elapsed time for each disk access. The efficiency shown in the graphs is the total number of bytes transferred divided by the total elapsed time. Figure 29 shows how the speedup changes as the fraction of dormant data increases for various amounts of temperature drift, which was injected by adding a uniformly distributed component to the file-access table used by the second driver phase. There is at least an



FCFS scheduler, Queue = 1

Figure 30: Efficiency Speedup for Simulation Experiment

11% speedup for any dormant fraction and any drift fraction, and a significant speedup for no drift and mostly dormant data.

Figure 30 shows the expected speedups obtained by simulating conditions similar to those expected during the benchmark for the same dormant and drift conditions and a sequentiality factor of 90%, which we observed during the experiments. The observed speedups are lower than the results predicted by the simulator because our naive clustering algorithm separates inode data from file data.

Figure 31 shows the speedup in disk response time for various fractions of dormant data and drifted temperatures. We calculated the average disk response time by dividing the total elapsed time for all I/Os by the number of I/Os. Speedup appears to be unaffected by the drift.

The speedups shown in Figures 29 and 31 are due to decreased seek times alone. Since iPcress accesses the disk in all block sizes between 512 bytes and 32 Kbytes, we must examine the speedup for all block sizes. The average seek time for the RZ55 is 16 ms, the average rotational delay is 8.3 ms, and the average transfer times vary between 0.5 ms for a 512-byte block and 29.5 ms for a 32-Kbyte block. For an "average" block of 8 Kbytes,

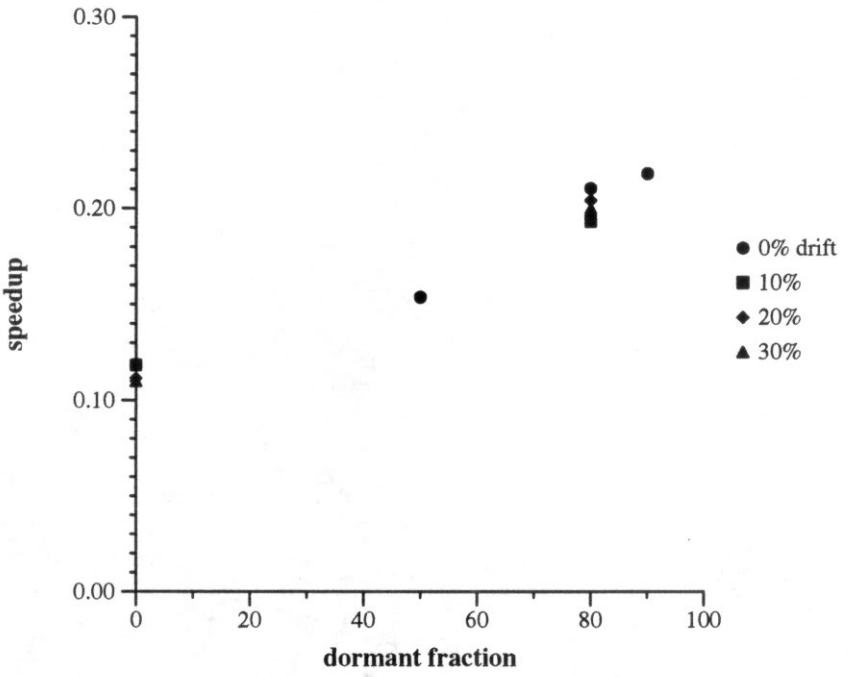


Figure 31: Speedup in Average Time per I/O

the transfer time is 7.4 ms, so the seek time accounts for only 50% of the total time of $16+8.3+7.4 = 31.7$ ms. We can interpret the average speedups shown in Figure 31 as being roughly one half of the average improvement in seek times. That is, a 10% improvement (e.g., 3.17 ms) suggests a 20% reduction (e.g., 3.17/16) in seek times. As the dormant fraction increases, the performance benefits of reorganizing data improve, roughly doubling by the time 90% of the data is dormant.

The disparity between the speedups shown in Figures 29 and 31 is due to changes in the distribution of access sizes between the first and second driver phases. During the second driver phase, iPcress accessed more small blocks, reducing both the average access time and throughput. This effect increases the speedups in Figure 31 and decreases the speedups in Figure 29. This shift in the access-size distribution was present only for non-zero drifts.

The time needed to reorganize the data is an important figure for real installations; iPcress reorganizes the 210 Mbyte file system in 4 hours. Reorganization starts with files distributed randomly; in practice, many files would already be in their correct locations from previous reorganizations.

Chapter 8

Conclusions and Future Work

iPcress demonstrates how to construct file systems that can improve performance by adaptively managing both user and system data. iPcress has two unique features that give it flexibility in optimizing all disk accesses: meta-data that is contained in files, and independent caching and storage strategies that can be selected dynamically on a per-file basis.

Chapter 7 shows that clustering is an effective means for improving disk performance, and it should be more effective for optical disks, since seek delay is a larger fraction of the total access time.

We have presented and analyzed only clustering, but there are other smart optimizations that deserve investigation. There are other file-access patterns that are both useful and detectable. Since files tend to be either read-only or write-only, a file system might use different storage and access methods for each. In particular, throughput and response time for write-only files might be improved by storing their data in a circular log and migrating the data out of the log only when necessary as done by LFS.

For read-only files, there are a variety of possible strategies depending on the file size and access characteristics (e.g., sequential or random). For small files, optimistic prefetching when the file is opened is likely to improve performance. For larger files that are read sequentially, performance might be improved by prefetching one or more blocks when the file is opened, and using a sliding window, prefectching data in front of the scan and flushing it behind.

Meta-data access characteristics have been largely ignored in research on file-access patterns. Meta-data access accounts for a significant fraction of file-system activity, so optimizing its access is important. One strategy might cluster all the active inodes together

near the center of the disk and spread the remaining (inactive) inodes over the disk near their files. Another strategy might place all inodes for a directory into a "bin" and then rearrange these bins according to the collective inode activity of each bin, placing the active bins near the center. Before these and other strategies can be evaluated, there must be some experimental research on inode access patterns.

Another area for future exploration is the interaction between prioritized or real-time disk scheduling and file-system activity. In particular, how could file systems use such disk schedulers to improve performance? Priority-based scheduling combined with optimistic prefetching and delayed writes might improve performance during peak loads by servicing the critical accesses immediately and delaying others until activity bursts subside. Also, the file system could place all dirty blocks on the disk queue at low priority so the disk could flush them with maximal throughput during idle periods.

One area of increasing importance is improving file system performance for heterogeneous devices. In particular, automatic migration of files between devices, such as between disk and electronic disk, or between disk and optical jukebox, will become crucial to system performance. Smart file systems will be able to use a combination of strategies, such as caching data from optical jukeboxes on disk, clustering active data on a single disk within a jukebox, or placing small active files on electronic disk and large active files on a RAID, to improve system performance.

Bibliography

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. MACH: a new kernel foundation for UNIX development. In *Proceedings USENIX 1986 Summer Conference*, pages 93–112, Atlanta, June 1986.

[2] E. R. Arnold and M. E. Nelson. Automatic UNIX backup in a mass-storage environment. In *Proceedings Winter 1988 USENIX*, pages 131–136, Dallas, Feb. 1988.

[3] H. P. Artis. Predicting the behavior of secondary storage management systems for IBM computer systems. In F. J. Kylstra, editor, *Performance '81*, pages 435–443. North-Holland Publishing Company, 1981.

[4] M. J. Bach. *The Design of the UNIX Operating System.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1986.

[5] M. J. Bach, M. W. Luppi, A. S. Melamed, and K. Yueh. A remote-file cache for RFS. In *Proceedings Summer 1987 USENIX*, pages 273–279, Phoenix, 1987.

[6] K. Baclawski. A stochastic model of data access. College of Computer Science, Northeastern University, Boston, Apr. 1987.

[7] K. Baclawski. A stochastic model of data access and communication. *Advances in Applied Mathematics*, 10(2):175–200, 1989.

[8] K. Baclawski, S. H. Contractor, and R. Wilmot. File access patterns: Self-similarity and time evolution. Technical Report NU-CCS-89-19, College of Computer Science, Northeastern University, Boston, 1989.

[9] M. G. Baker. DASD tuning – understanding the basics. In *Proceedings International Conference on Management and Performance Evaluation of Computer Systems*, pages 1251–1257, Reno, NV, Dec. 1989.

[10] R. E. Barkley and T. P. Lee. A dynamic file system inode allocation and reclaim policy. In *Proceedings Winter 1990 USENIX*, pages 1–9, Washington, DC, Jan. 1990.

[11] R. Baron, R. Rashid, E. Siegel, A. Tevanian, and M. Young. MACH-1: an operating environment for large-scale multiprocessor applications. *IEEE Software*, 2(4):65–67, July 1985.

[12] A. L. Bastian. Cached DASD performance prediction and validation. In *Proceedings of the CMG 13th International Conference*, pages 174–177, Dec. 1982.

[13] A. L. Bastian, J. S. Hyde, and W. E. Langstroth. Characteristics of DASD use. In *Proceedings of the CMG 12th International Conference*, pages 107–109, Phoenix, Dec. 1981.

[14] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, 1987.

[15] P. Biswas and K. K. Ramakrishnan. File access characterization of VAX/VMS environments. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 227–234, Paris, May 1990.

[16] D. Bitton and J. Gray. Disk shadowing. In *Proceedings of the 14th VLDB Conference*, pages 331–338, Los Angeles, 1988.

[17] M. Blaze and R. Alonso. Long-term caching strategies for very large distributed file systems. In *Proceedings USENIX Summer 1991 Conference*, pages 3–15, Nashville, June 1991.

[18] A. Brandwajn. Aspects of DASD performance. *Systems Performance Architecture Journal*, 4(2):29–52, May 1984.

[19] L.-F. Cabrera and D. D. E. Long. Swift: a storage architecture for very large objects. Technical Report RJ7128(67375), Computer Science, IBM Almaden Research Center, Almaden, CA, Nov. 1989.

[20] S. D. Carson and P. Vongsathorn. Error bounds on disk arrangement using frequency information. *Information Processing Letters*, 31(4):209–213, May 1989.

[21] V. Cate and T. Gross. Combining the concepts of compression and caching for a two-level filesystem. In *Proceedings Architectural Support for Programming Languages and Operating Systems*, pages 200–211, Santa Clara, CA, Apr. 1991.

[22] A. Chang, M. Mergen, S. Porter, B. Rader, and J. Roberts. Evolution of storage facilities in the AIX operating system. In *IBM RISC System/6000 Technology*, pages 138–143. IBM, 1990.

[23] E. E. Chang. *Effective Clustering and Buffering in an Object-Oriented DBMS.* PhD thesis, Computer Science Division, University of California, Berkeley, June 1989.

[24] P. Chen. An evaluation of redundant arrays of disks using an Amdahl 5890. Technical Report UCB/CSD 89/506, Computer Science Division, University of California, Berkeley, May 1989.

[25] P. Chen, G. Gibson, R. H. Katz, D. A. Patterson, and M. Schulze. Two papers on RAIDs. Technical Report UCB/CSD 88/479, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, Dec. 1988.

[26] G. Copeland, T. Keller, R. Krishnamurthy, and M. Smith. The case for safe RAM. In *Proceedings of the 15th VLDB Conference*, pages 327–335, Amsterdam, Aug. 1989.

[27] D. W. Cornell and P. S. Yu. Integration of buffer management and query optimization in relational datase environment. In *Proceedings of the 15th VLDB Conference*, pages 247–255, Amsterdam, Aug. 1989.

[28] C. J. Date. *An Introduction to Database Systems*, volume 2. Addison-Wesley, Reading, MA, 1985.

[29] C. J. Date. *An Introduction to Database Systems, (4th Edition)*, volume 1. Addison-Wesley, Reading, MA, 1985.

[30] L. Davis. Personal communication, Oct. 1990.

[31] P. J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, May 1968.

[32] P. J. Denning. Working sets past and present. *IEEE Transactions on Software Engineering*, SE-6(1):64–84, Jan. 1980.

[33] M. Devarakonda and R. K. Iyer. A user-oriented analysis of file usage in UNIX. In *Proceedings of COMPSAC '86*, pages 21–27, Chicago, Oct. 1986.

[34] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. In *Proceedings SIGMOD*, pages 1–8, Boston, June 1984.

[35] P. C. Dibble. *A Parallel Interleaved File System.* PhD thesis, Department of Computer Science, University of Rochester, Rochester, NY, Mar. 1990.

[36] Digital Equipment Corporation. *Information and Configuration Guide for Digital's Desktop Storage: Focus on the RZ Series of SCSI Disk Drives*, May 1990.

[37] J. Dixon, G. Marazas, A. McNeill, and G. Merckel. Dynamic control of I/O channel utilization in a cached disk system. *IBM Technical Disclosure Bulletin*, 26(1):107–108, June 1983.

[38] M. C. Easton. Computation of cold-start miss ratios. *IEEE Transactions on Computers*, C-27(5):404–408, May 1978.

[39] R. S. Finlayson and D. Cheriton. Log files: an extended file service exploiting write-once storage. *ACM Operating System Review*, 21(5):139–148, Nov. 1987.

[40] A. Flory, J. Gunther, and J. Kouloumdjian. Data base reorganization by clustering methods. *Information Systems*, 3(1):59–62, 1978.

[41] R. Floyd. Directory reference patterns in a UNIX environment. Technical Report TR-179, Computer Science Department, University of Rochester, Rochester, NY, Aug. 1986.

[42] R. Floyd. Short-term file reference patterns in a UNIX environment. Technical Report TR-177, Computer Science Department, University of Rochester, Rochester, NY, Mar. 1986.

[43] R. Floyd. *Transparency in Distributed File Systems.* PhD thesis, Department of Computer Science, University of Rochester, Rochester, NY, Jan. 1989.

[44] H. Frank. Analysis and optimization of disk storage devices for time-sharing systems. *Journal of the ACM*, 16(4):602–620, Oct. 1969.

[45] M. Fridrich and W. Older. The FELIX file server. In *Proceedings 8th Symposium on Operating System Principles*, pages 37–44, Pacific Grove, CA, Dec. 1981.

[46] M. B. Friedman. DASD access patterns. In *Proceedings of the 1983 CMG International Conference*, pages 51–61, Dec. 1983.

[47] S. H. Fuller. Minimal-total-processing time drum and disk scheduling disciplines. *Communications of the ACM*, 17(7):376–381, July 1974.

[48] J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 1970(2):78–117, 1970.

[49] R. Geist and S. Daniel. A continuum of disk scheduling algorithms. *ACM Transactions on Computer Systems*, 5(1):77–92, Feb. 1987.

[50] G. Gibson. Performance and reliability in redundant arrays of inexpensive disks. Technical Report UCB/CSD 89/535, Computer Science Division, University of California, Berkeley, Sept. 1989.

[51] G. Gibson, L. Hellerstein, R. Karp, R. Katz, and D. Patterson. Coding techniques for handling failures in large disk arrays. Technical Report UCB/CSD 88/477, Computer Science Division, University of California, Berkeley, Dec. 1988.

[52] G. Gibson, L. Hellerstein, R. Karp, R. Katz, and D. Patterson. Failure correction techniques for large disk arrays. In *Proceedings Architectural Support for Programming Languages and Operating Systems*, pages 123–132, Boston, Apr. 1989.

[53] D. Gifford and A. Spector. The TWA reservation system. *Communications of the ACM*, 27(7):649–665, July 1984.

[54] C. G. Gray and D. R. Cheriton. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings Twelfth ACM Symposium on Operating Systems*, pages 202–210, Litchfield Park, AZ, Dec. 1989.

[55] J. Gray. Why do computers stop and what can be done about it? Technical Report 85.7, Tandem Computers, Cupertino, CA, June 1985.

[56] J. Gray. A census of Tandem system availability between 1985 and 1990. Technical Report 90.1, Tandem Computers, Cupertino, CA, Jan. 1990.

[57] C. P. Grossman. Cache-DASD storage design for improving system performance. *IBM Systems Journal*, 24(3/4):316–334, 1985.

[58] D. D. Grossman and H. F. Silverman. Placement of records on a secondary storage device to minimize access time. *Journal of the ACM*, 20(3):429–438, July 1973.

[59] R. Gusella. The analysis of diskless workstation traffic on an ethernet. Technical Report UCB/CSD 87/379, Computer Science Division (EECS), University of California, Berkeley, Nov. 1987.

[60] J. L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, May 1988.

[61] R. G. Guy, J. S. Heidemann, W. Mak, J. Thomas W. Page, G. J. Popek, and D. Rothheimer. Implementation of the Ficus replicated file system. In *Proceedings USENIX 1990 Summer Conference*, pages 63–71, Anaheim, CA, June 1990.

[62] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–316, Dec. 1983.

[63] R. Hagmann. Reimplementing the cedar file system using logging and group commit. *ACM Operating Systems Review*, 21(5):155–162, Nov. 1987.

[64] D. Hendricks. A filesystem for software development. In *Proceedings USENIX Summer 1990 Conference*, pages 333–340, Anaheim, CA, June 1990.

[65] M. Henley and I. Y. Liu. Static vs dynamic management of consistently very active data sets. In *International Conference on Management and Performance Evaluation of Computer Systems*, pages 208–216, Orlando, Dec. 1987.

[66] J. H. Howard. An overview of the Andrew file system. In *Proceedings USENIX Winter 1988 Conference*, pages 23–26, Dallas, Feb. 1988.

[67] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayan, R. N. Sidebottom, and M. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, Feb. 1988.

[68] IBM. Transaction processing facility. Order Number GH20-7488-00, IBM, June 1989.

[69] D. M. Jacobson. Serial vs. random file access patterns in various workloads. Technical Report HPL-CSP-90-43, Hewlett-Packard Laboratories, Concurrent Computing Department, Palo Alto, CA, Dec. 1990.

[70] D. M. Jacobson and J. Wilkes. Disk scheduling algorithms based on rotational position. Technical Report HPL-CSP-91-7, Hewlett-Packard Laboratories, Concurrent Computing Department, Palo Alto, CA, Feb. 1991.

[71] R. H. Katz, G. Gibson, and D. A. Patterson. Disk system architectures for high performance computing. Technical Report UCB/CSD 89/497, Computer Science Division, University of California, Berkeley, Mar. 1989.

[72] M. L. Kazar. Synchronization and caching issues in the Andrew file system. In *Proceedings Winter 1988 USENIX*, pages 27–36, Dallas, Feb. 1988.

[73] M. L. Kazar and et al. DEcorum file system architectural overview. In *Proceedings Summer 1990 USENIX*, pages 151–163, Anaheim, CA, June 1990.

[74] B. E. Keith. Perspectives on NFS file server performance characterization. In *Proceedings Summer 1990 USENIX Conference*, pages 267–277, Anaheim, CA, June 1990.

[75] S. Kleiman. Vnodes: an architecture for multiple file system types in Sun UNIX. In *Proceedings Winter 1986 USENIX Conference*, pages 238–247, Atlanta, Jan. 1986.

[76] D. E. Knuth. *Fundamental Algorithms*, volume 1. Addison-Wesley, Reading, MA, 1973.

[77] D. E. Knuth. *Searching and Sorting*, volume 3. Addison-Wesley, Reading, MA, 1973.

[78] P. D. L. Koch. Disk file allocation based on the buddy system. *ACM Transactions on Computer Systems*, 5(4):352–370, Nov. 1987.

[79] J. F. Kurose and R. Simha. A microeconomic approach to optimal file allocation. In *Proceedings 6th International Conference on Distibuted Computing Systems*, pages 28–35, Cambridge, MA, May 1986.

[80] E. D. Lazowska and J. Zahorjan. File access performance of diskless workstations. *ACM Transactions on Computer Systems*, 4(3):238–268, Aug. 1986.

[81] M. K. McKusik, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, Aug. 1985.

[82] J. M. Mott and J. C. O'Quin. Auxiliary storage management in the AIX operating system. In *IBM RISC System/6000 Technology*, pages 144–149. IBM, 1990.

[83] M. N. Nelson, B. B. Welch, and J. K. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–154, Feb. 1988.

[84] J. Ousterhout and F. Douglis. Beating the I/O bottleneck: a case for log-structured file systems. *ACM Operating Systems Review*, 23(1):11–28, Jan. 1989.

[85] J. K. Ousterhout, H. D. Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson. A trace-driven analysis of the UNIX 4.2 BSD file system. In *Proceedings of the 10th ACM Symposium on Operating System Principles*, pages 15–24, Orcas Island, WA, Dec. 1985.

[86] A. Park, J. C. Becker, and R. J. Lipton. IOStone: a synthetic file system performance benchmark. *ACM Computer Architecture News*, 18(2):45–52, June 1990.

[87] K. R. Pattipati, J. Wolf, and S. Deb. A calculus of variations approach to file allocation problems in computer systems. Technical Report RC14948(66816), IBM Research Division, Sept. 1989.

[88] J. K. Peacock. The counterpoint fast file system. In *Proceedings Winter 1988 USENIX*, pages 243–249, Dallas, Feb. 1988.

[89] B. R. Pierce. An MVS SRM discussion. Technical Bulletin GG66-0201-00, IBM Washington Systems Center, Apr. 1985.

[90] E. Rahm and D. Ferguson. Cache management algorithms for sequential data access. Technical Report RC15486(68889), IBM Research Division, T.J. Watson Research Center, Yorktown Heights, NY, Feb. 1990.

[91] D. Ritchie and K. Thompson. UNIX time-sharing system. *Bell System Technical Journal*, 57(6):1905–1929, July 1978.

[92] D. M. Ritchie. Personal communication, Apr. 1991.

[93] J. T. Robinson and M. V. Devarakonda. Data cache management using frequency-based replacement. In *Proceedings ACM Sigmetrics*, pages 134–142, Boulder, CO, May 1990.

[94] M. Rosenblum and J. K. Ousterhout. The LFS storage manager. In *Proceedings Summer 1990 USENIX*, pages 315–324, Anaheim, CA, June 1990.

[95] C. Ruemmler. Shuffleboard — methods for adaptive data reorganization. Technical Report HPL-CSP-90-41, Hewlett-Packard Laboratories, Concurrent Computing Department, Palo Alto, CA, Aug. 1990.

[96] K. Salem and H. Garcia-Molina. Disk striping. In *Proceedings International. Conference on Data Engineering*, pages 336–342, Los Angeles, Feb. 1986.

[97] M. Satyanarayanan. A study of file sizes and functional lifetimes. In *Proceedings 8th Symposium on Operating System Principles*, pages 96–108, Pacific Grove, CA, Dec. 1981.

[98] M. Satyanarayanan. A survey of distributed file systems. Technical Report CMU-CS-89-116, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1989.

[99] B. Schneiderman. Optimum data base reorganization points. *Communications of the ACM*, 16(6):362–365, June 1973.

[100] M. D. Schroeder, D. K. Gifford, and R. M. Needham. A caching file system for a programmer's workstation. *ACM Operating Systems Review*, 19(5):25–34, Dec. 1985.

[101] M. E. Schulze. Considerations in the design of a RAID prototype. Technical Report UCB/CSD 88/448, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, Aug. 1988.

[102] M. Seltzer, P. Chen, and J. Ousterhout. Disk scheduling revisited. In *Proceedings Winter 1990 USENIX*, pages 313–323, Washington, D.C., Jan. 1990.

[103] A. Siegel, K. Birman, and K. Marzullo. Deceit: a flexible distributed file system. Technical Report 89-1042, Department of Computer Science, Cornell University, Ithaca, NY, Nov. 1989.

[104] D. P. Siewiorek and R. S. Swarz. *The Theory and Practice of Reliable System Design.* Digital Press, Bedford, MA, 1982.

[105] A. Sikeler. Var-page-LRU: a buffer replacement algorithm supporting different page sizes. In G. Goos and J. Hartmanis, editors, *Lecture Notes in Computer Science 303*, pages 336–351. Springer Verlag, 1988.

[106] A. J. Smith. Sequentiality and prefetching in database systems. *ACM Transactions on Database Systems*, 3(3):223–247, Sept. 1978.

[107] A. J. Smith. Analysis of long term file reference patterns for application to file migration algorithms. *IEEE Transactions on Software Engineering*, SE-7(4):403–416, July 1981.

[108] A. J. Smith. Optimization of I/O systems by cache disks and file migration: a summary. *Performance Evaluation*, 1(3):249–262, Nov. 1981.

[109] A. J. Smith. Disk cache - miss ratio analysis and design considerations. *ACM Transactions on Computer Systems*, 3(3):161–203, Sept. 1985.

[110] C. Staelin. File access patterns. Technical Report CS-TR-179-88, Department of Computer Science, Princeton University, Princeton, NJ, Sept. 1988.

[111] C. Staelin and H. Garcia-Molina. Clustering active disk data to improve disk performance. Technical Report CS-TR-283-90, Department of Computer Science, Princeton University, Princeton, NJ, Sept. 1990.

[112] C. Staelin and H. Garcia-Molina. File system design using large memories. In *Proceedings IEEE Jerusalem Conference*, pages 11–21, Jersusalem, Oct. 1990.

[113] C. Staelin and H. Garcia-Molina. Smart filesystems. In *Proceedings Winter 1991 USENIX*, pages 47–54, Dallas, Jan. 1991.

[114] A. Tevanian and R. Rashid. MACH: a basis for future UNIX development. Technical Report CMU-CS-87-139, Carnegie-Mellon University, June 1987.

[115] J. Thisquen. 6880 cache controller feature and 6680 electronic direct access storage planning and tuning guide. Publication Number MT-A01077-003, Amdahl Corp., Sunnyvale, CA, 1986.

[116] R. van Renesse. *The Functional Processing Model.* PhD thesis, Vrije Universiteit, Amsterdam, 1989.

[117] P. Vongsathorn and S. D. Carson. A system for adaptive disk rearrangement. *Software — Practice and Experience*, 20(3):225–242, Mar. 1990.

[118] J. Wilkes and R. Stata. Specifying data availability in multi-device file systems. *ACM Operating Systems Review*, 25(1):56–59, Jan. 1991.

[119] R. Wilmot. File usage patterns from SMF data: Highly skewed usage. In *Proceedings International Conference on Management and Performance Evaluation of Computer Systems*, pages 668–677, Reno, NV, Dec. 1989.

[120] J. Wolf. The placement optimization program: a practical solution to the DASD file assignment problem. *Performance Evalution Review*, 17(1):1–10, May 1989.

[121] B. L. Wolman and T. M. Olson. IOBENCH: a system independent IO benchmark. *ACM Computer Architecture News*, 17(5):55–70, Sept. 1989.

[122] P. C. Yue and C. K. Wong. On the optimality of the probability ranking scheme in storage applications. *Journal of the ACM*, 20(4):624–633, Oct. 1973.