MIT/LCS/TR–528

FILE SYSTEMS WITH MULTIPLE FILE IMPLEMENTATIONS

Raymie Stata

February 1992

File Systems with Multiple File Implementations

by

Raymie Stata

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Electrical Engineering and Computer Science

and

Master of Science in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

May 1990

Signature of Author ....
Department of Electrical Engineering and Computer Science
May 22, 1990

Certified by ....

Barbara Liskov

N.E.C. Professor of Software Science and Engineering

Thesis Supervisor

Certified by ....

John Wilkes

Project Manager, Hewlett-Packard Laboratories

Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

File Systems with Multiple File Implementations

by

Raymie Stata

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 1990, in partial fulfillment of the
requirements for the degrees of

Bachelor of Science and Master of Science

in Electrical Engineering and Computer Science

Abstract

This thesis proposes ideas for designing file system software for large, high-performance file server hardware we feel will be common in the middle to late nineties. In particular, the thesis examines the value and pragmatics of file systems with multiple file implementations. A file implementation determines how a file is represented in secondary storage and the procedures by which that representation is interpreted. A file system with multiple file implementations can use different implementations for different files. The thesis also proposes an allocation algorithm designed for a system with device parallelism and multiple file implementations, and it reports the results of a trace-driven simulation study evaluating the algorithm. The thesis proposes parameterizing file behavior to give users control over which implementation is used for a file without exposing the low-level details of implementations.

Thesis Supervisor: Barbara Liskov
Title: N.E.C. Professor of Software Science and Engineering

Thesis Supervisor: John Wilkes
Title: Project Manager, Hewlett-Packard Laboratories

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A curiosity just ten years ago, networks of single-user computers are now common. And as computer networks have become more common, so have the file servers that serve them. Both hardware and software for file servers have advanced rapidly in the last ten years, and the progress continues.

This thesis proposes ideas for designing file system software for the large, high-performance file server hardware we feel will be common in the middle to late nineties. In particular, the thesis examines the value and pragmatics of file systems with multiple file implementations. A file implementation determines how a file is represented in secondary storage and the procedures by which that representation is interpreted. A typical file implementation breaks files into equal sized blocks and store each block in a contiguous region on a disk. A variant of this implementation compresses files before breaking them into blocks to save space. Another variant stores each block on two different disks to increase fault-tolerance. A file system with multiple file implementations can use different implementations for different files.

The next section describes our assumptions about file server hardware in the middle to late nineties. In brief, we assume that future file servers will have substantially more processing power than today. They will contain a large, heterogeneous set of storage devices in configurations that will allow concurrent access. The section following outlines the contents of the thesis and delimits its scope.

1.1 Future file servers

The ideas in this thesis are based on assumptions about the future of file servers. These assumptions are based on current trends in computer system hardware and in file server use patterns, including the growing I/O gap, the proliferation of storage media types, the increasing use of device parallelism, and the increasing overall scale of file servers. This section discusses these trends and projects a vision of future file servers based on this discussion.

1.1.1 The growing I/O gap

A significant trend faced by file server designers is the growing I/O gap. CPU performance, network bandwidth, and primary and secondary storage density are increasing exponentially, but the access times and transfer rates of secondary storage are failing to keep pace [Ousterhout89, Wilkes90]. In fact, the access time to newer media such as magneto-optical disk is longer than preceding technology. This "I/O gap" is aggravated by other factors. As computer networks become increasingly common, file servers must handle the workloads of increasing numbers of systems. Also, growing use of digitized sound and video is putting heavy demands on file server throughput and capacity.

Closing the I/O gap is perhaps the largest problem faced by designers of file servers. Fortunately, the trends behind the I/O gap contain the seeds to a solution. First, as the price of CPU performance drops, designers can put more processing power into the server. Increased processing power will help close the I/O gap by allowing the use of sophisticated software that aggressively optimizes file system performance. Second, as the density of primary storage increases, designers can use larger file caches, both at the server and the client. Large file caches have already proven effective, and researchers are calling for even larger caches in the future [Ousterhout89].

1.1.2 The proliferation of storage media

Another trend faced by file system designers is the proliferation of secondary storage media types. Ten years ago, most files were stored on two media: magnetic disk and tape. Today, files are stored on more types of media, including flash RAM, CD-ROM, optical WORM, videotape, magneto-optical disk, and digital audio tape (DAT). Additionally, since file servers represent long-term investments, a mature server is likely to contain several generations of the same medium. Given the rapid advances in technology, latter generations are likely to have important differences from their earlier counter-parts.

Each new medium has its own pattern of optimal access and challenges file system designers to use it according to its own idiosyncrasies. Since many of the new media types are both denser and slower than magnetic disk, one result of the new media will be deeper storage hierarchies. Another result will be a number of specialized file implementations, such as log-structured file systems for optical WORM [Finlayson87].

1.1.3 The increasing use of device parallelism

A third trend faced by file system designers is the increasing use of storage device parallelism. N-dimensional disk arrays are a configuration of disks with N independent I/O channels used to increase bandwidth. Recently, researchers in the workstation world have been looking at disk arrays to both increase bandwidth and decrease latency in hopes of closing the I/O gap [Patterson88, Livny87].

Although disk arrays have proven successful in bettering performance, they have also increased demand for better fault-tolerance. Increasing the number of disks on which a file is stored in order to increase potential parallelism also increases the probability that the file will fail. The more devices a file is striped across, the higher the probability that one of them will fail [Schulze89]. Although the failure rates of secondary storage are decreasing dramatically [Gray90], relying on the reliability of individual devices is not a cost-effective solution to the fault-tolerance problem of disk arrays; rather, redundancy is needed [Patterson88].

1.1.4 The increasing demand for capacity

The final trend we will consider is the growing size of file servers. If past history is any indication, then as CPU performance continues to grow exponentially, demand for secondary storage capacity will grow with it. This trend is aggravated by the fact that servers are being used by increasing numbers of clients. It is further aggravated by applications using digital audio and video, which require substantial secondary storage capacity. Finally, the trend towards disk arrays and the proliferation of storage media both increase the number of devices in servers.

This trend introduces a new challenge for file system designers: hide the existence of individual devices from users and administrators. Make the system configuration transparent. Automate administrative functions, including failure recovery (fault-tolerance), hierarchy management (archiving), and configuration management. Given the use of disk arrays, automatic failure recovery is especially important.

1.1.5 Future file servers

Based on these trends, one vision of where file servers will be in the future is as follows. Future file servers will be built from a large, heterogeneous set of devices, and they will have much more RAM and processing power than today's servers. They will overcome the I/O gap by using caching, device parallelism, and sophisticated software optimizations. They will perform automatically most administrative functions, in particular failure recovery.

An example of such a file server is the DataMesh, a large, high-performance file server being developed at Hewlett-Packard Laboratories [Wilkes89a, Wilkes89b]. The DataMesh is a large file server (.1–1.5 terabytes) built from a tightly coupled network of DataMesh nodes. Each DataMesh node has a secondary storage device (e.g., a hard disk) and a 20 MIPS processor.

The ideas for file server software proposed in this thesis were developed specifically for the DataMesh. However, the ideas should be applicable to any file server fitting the overall vision described above, including ones built around a very fast uniprocessor.

1.2 Future file systems for servers

This thesis is concerned with file system software for the type of file servers described in the previous section. It presents ideas that may prove useful for building such software. The ideas remain largely speculative. Neither an implementation nor a detailed simulation were built; rather, the ideas are supported by work reported in the literature, measurements of existing machines, and first-order simulations.

Chapter 2 examines the larger issues surrounding the file systems we are interested in. The chapter focuses on file implementations. It argues that file systems for future servers should be built with multiple file implementations, a theme that unifies the thesis. The chapter discusses problems faced when building such systems and sketches a design for a file system that solves these problems.

Chapters 3 and 4 examine in more detail two parts of the file system proposed in Chapter 2. Chapter 3 looks at allocation of secondary storage, in particular, how to find physical disk locations for file blocks. It proposes an allocation algorithm designed for a system with device parallelism. It reports the results of a simple, trace-driven simulation study evaluating the algorithm. It argues that the algorithm is well suited to a file system with multiple file implementations.

Chapter 4 describes parameters for file behavior. Users will set these parameters according to their needs, and the system will use these parameters to assign files to file implementations. The parameters are designed to give users control over which implementation is used for a file without exposing the low-level details of implementations.

Chapter 5 summarizes the ideas presented in the thesis and discusses their potential benefits.

Chapter 2

File Systems with Multiple File Implementations

This chapter outlines a proposed design for file system software for file servers of the type described in the introduction. The chapter focuses on file implementations, an important concept in the design. Section 2.1 defines file implementations and reviews at a few implementations proposed in the literature. Section 2.2 argues that future servers should be built with multiple implementations. Section 2.3 describes Sun Microsystems's Virtual File System (VFS)—an existing file system with multiple file implementations—and discusses its weaknesses. Section 2.4 describes the proposed file system itself. Section 2.5 summarizes the ideas in this chapter and looks ahead to the next chapters.

2.1 File implementations

A "file implementation" is the part of the file system software that determines the representation of files in secondary storage and the procedures by which that representation is interpreted.

No widely accepted term has come to refer to file implementations, but researchers have not ignored them. In fact, file implementations have been the subject of much research recently, work that motivated the ideas in this chapter. Some of this work is reviewed below. The review is not exhaustive, but it should serve to sharpen the meaning of "file implementation" and to point out tradeoffs in the design of file implementations. These tradeoffs are the basis of the argument in the next section for building file systems with multiple implementations.

2.1.1 Mirrored file implementations

Disk mirroring is probably the oldest and most common "alternative" file implementation. Disk mirroring increases fault-tolerance by keeping two copies of each file on independent disks. If one copy is lost due to media failure, the second copy will still be available. Disk mirroring implementations can increase read performance by reading from both copies concurrently, which works particularly well for workloads with a heavy read bias. Disk mirroring is relatively common for large database systems; Tandem, IBM, HP, and other companies offer mirroring in their systems.

2.1.2 Striping file implementations

File striping is a file implementation that takes advantage of storage device parallelism. These implementations spread, or "stripe," the data of a single file across multiple devices (the number of devices is called the stripe width). Files can be striped at the bit level [Tucker88, Ng89], at the byte level [Kim85], or at the block level [Livny87, Patterson88, Henderson89]. File striping can potentially multiply throughput by the stripe width [Patterson88]. File striping can also decrease queuing time by spreading workload across disks [Livny87]. File striping has proven successful in practice for very large files that are accessed sequentially [Henderson89]. However, it is not clear how well striping works for small files [Ousterhout89].

As mentioned in the introduction, striping a file devices greatly decreases the mean time to failure (MTTF) for a file since the MTTF of N disks failing is $1/N$ that of a single disk (assuming independent failures). As a result, file striping is usually joined with redundancy for fault-tolerance. Patterson *et. al.* survey a range of redundancy schemes for file striping [Patterson88]. The most popular of these options is a single parity stripe: given a stripe width of N disks, each stripe consists of $N-1$ data blocks and one parity block. Because the MTTF of this configuration is roughly the square of the MTTF of a single disk divided by N (the number of disks in a stripe), this configuration is reliable enough for all but the largest stripes or most unreliable disks.

2.1.3 Log-structured file implementations

The log-structured file system (LFS) is another implementation that takes advantage of storage device parallelism [Ousterhout89, Rosenblum90]. In the office and engineering environments, files tend to be small, often less than 8 kilobytes, which limits value of striping [Satyanarayanan81, Ousterhout85]. The LFS organizes many small files into one long, sequentially written log which in turn can be striped to great advantage.

The designers of the LFS assume that as file-caches increase in size, hit rates will also increase; eventually, most reads will be satisfied by the cache, and disk traffic will be mostly writes. The LFS combines changes to files into a log-like structure which it writes to disk in large, sequential chunks. In essence, the LFS transforms the small, random transfers seen at the logical level into large, sequential transfers at the physical level, a workload well-suited to disk arrays.

(This workload is also well-suited to single disks. However, simulation studies indicate that under most conditions declustering data across multiple disks performs better than clustering the data on a single disk [Livny87]. Given our assumption that future file servers will have multiple devices available, log-structured file implementations will most likely be striped.)

2.1.4 Delayed-write file implementations

The delayed-write file implementation trades-off fault-tolerance for performance [Ohta90]. UNIX[1] uses a write-back cache to speed file accesses. To minimize data loss in the case of a failure, UNIX makes two exceptions to the write-back discipline. First, it periodically flushes all dirty buffers, usually every half-minute. Second, it writes-through critical file system structures to maintain important on-disk invariants. The delayed-write file implementation eliminates the periodic cache flush and writes all data asynchronously.

This "fast-and-loose" file system trades off reliability for speed. High reliability is not a concern for all files, e.g., compiler intermediate files. Ohta and Tezuka designed the delayed-write file system implementation specifically for the `/tmp` directory, which usually contains such temporary files.

2.2 Multiple file implementations

Multiple implementations will help close the I/O gap. One way they will do this is by providing a mechanism to aggressively manage trade-offs. File implementations are specialized. Different implementations use different amounts of storage and require different amounts of processor time. Different implementations are optimized for different file sizes and different access patterns, and they provide different levels of fault-tolerance. The design space for file implementations is large and complex; tradeoff variables include read and write performance, reliability, space efficiency, file access patterns, and file sizes.

Large file servers of the future will service a heterogeneous workload, ranging from small word processing files to large, real-time video files, and no single file implementation is likely to provide adequately for the entire range. Instead, file servers will need a number of complementary file implementations working together to provide better tuned service. (Although, if the I/O gap is closed by storage technology such as holographic storage [Parish90], going back to "one-size-fits-all" file systems might be possible.)

[1] UNIX is a trademark of AT&T, Inc.

An example will make this argument more concrete. Parity striping requires less redundant storage than disk mirroring but requires an extra disk access for small writes [Patterson88]. A file system with multiple implementations could take advantage of the strengths of both by using mirroring when small writes are expected and parity striping when large writes (or a heavy read bias) are expected.

Another way multiple file implementations will help close the I/O gap is by providing a mechanism to focus the increased processing power of future servers. For example, a file implementation could compress the file before sending it to disk [Cate90] or derive a file from a set of parameters.

2.3 The Virtual File System

File systems with multiple file implementations are already being built and sold. For example, in addition to the BSD 4.3 file implementation, Hewlett-Packard's UNIX offers a disk mirroring file implementation, a delayed write file implementation, and a special file implementation for their CD-ROM product. At DEC, an experimental UNIX system was built that can access media from MS-DOS, VMS, and BSD 4.3 [Koehler87].

UNIX file systems with multiple implementations are based on Sun Microsystem's VFS. (Variations on VFS have been proposed and built, e.g., see [Rodriguez86] and [Karels86].) Although VFS was introduced to allow remote file access, its design is general. It provides a general mechanism for multiple implementations; remote access is achieved by suppling an implementation that implements file operations by remote procedure calls to a file server. This section will sketch the design of the multiple implementation mechanism in VFS and will examine the problems with it.

2.3.1 VFS overview

VFS is based on two basic concepts: volumes and the file system switch. A volume is a disk partition holding files organized in a tree-like naming structure. ("Volumes" are usually called “file systems;” to avoid confusion with our own use of “file system,” we have used “volume” instead.) The internal nodes of the tree structure are directories, and the leaves are files. A volume is “mounted” by making its root directory available as an internal node (directory) of a volume that is already mounted. The mounting process is boot-strapped by designating a special volume, called the “root,” to be mounted at boot time.

VFS uses the volume construct to assign files to file implementations. It associates each volume with a single file implementation; all files in the volume use this implementation.

The second important concept in VFS is the file system switch. File system switches are used to route file operations from the system call handler to the appropriate file implementation. A switch is a table of pointers to the functions that implement file operations such as read and write. A file's file system switch is stored in the file's *vnode*, a kernel structure holding important information about open files. When VFS has to execute a file operation like read, it finds the file's switch in the file's vnode, then finds the code for the operation in the switch. This approach has a definite “object-oriented” flavor and is very close to virtual tables in C++. The switch put in a file's vnode is determined by the volume holding the file.

2.3.2 Problems with VFS

VFS has problems caused by volumes, a construct that VFS inherited from Unix. VFS uses the volume construct for three functions. Volumes are used for naming: all files in the same volume share a common root directory. Volumes are used for disk selection: all files in the same volume are stored on the same disk. Volumes are used for implementation selection: all files in the same volume use the same file implementation. Combining naming with disk and implementation selection has drastic consequences.

Combining naming and disk selection interferes with load balancing. Table 2.1 illustrates the problem. On two different machines run by different administrators, disk

Table 2.1: Access distributions to disks on two machines

Disk no	Cello	Red
	Percent of accesses	
Disk 0	41%	41%
Disk 1	2%	15%
Disk 2	8%	22%
Disk 3	6%	7%
Disk 4	1%	1%
Disk 5	42%	14%

workloads are grossly unbalanced. (Section 3.2 will describe the machines in more detail). The problem is that systems have "directory locality:" files in certain directory sub-trees are accessed more often than those in other directories. In Table 2.1, for example, disk zero in both cases (the most heavily used disks) contained most of the heavily accessed `/usr` directory. By constraining large sub-trees to reside on a single disk, UNIX practically guarantees unbalanced disks unless tedious configuration precautions are taken.

Finally, combining naming and implementation selection interferes with the purpose of the name tree. The name tree should be used to express the logical relationships between files. The relationship between a file and its implementation is orthogonal to the file's logical relationships to other files. For example, when organizing the files used for the experiments reported in Chapter 3, we wanted the files in the same sub-tree. However, the files holding trace data were large, they had to be assigned to the file implementation that used a magneto-optical jukebox. As a result, the trace files ended up in a directory unrelated to the project's main directory. Symbolic links help mitigate this problem, but only by placing an extra burden on the user.

2.4 A proposed design for file systems

This section sketches a design for the file system software of file servers with a large, heterogeneous set of secondary storage devices. The file abstraction is central to file systems, and in this design the abstraction is implemented by multiple file implementations. Operations for a file are directed by a switch to the file's implementation. Files are assigned to implementations by the file assignment module. The design has two types of persistent storage. The design is illustrated in Figure 2-1. The database holds file

Figure 2-1: Design for file system software

system metadata such as file names, file attributes, and file locations. File block storage stores file blocks on various device types. The block placement module finds physical locations for file blocks. The type of device holding a file block is determined by the file's implementation; the particular device and the location within that device are determined by the block placement module (see Chapter 3).

An example of a file being created, written and read illustrates how these components work together. When a file is created, the file assignment module makes an entry for the file in the database and assigns the file to one of the file implementations (criteria for assignment will be discussed later). Write operations are handled by the file's file implementation. This implementation uses the block placement module to find free storage for the file, and it uses the database to record where the parts of the file are located. Read operations are also handled by the file's implementation, which uses the database to find parts of the file to be read. Access statistics such as the number and size of reads are kept for each file in the database; these statistics can be used by the file implementation for optimizations and by the file assigner to re-assign the file to a better implementation.

2.4.1 File implementations

A file implementation provides customized service on a per-file basis. It is a means of managing trade-offs to tune the file system to the static and dynamic characteristics of individual files. Thus, file implementations should include only those functions for which there exist significant trade-offs.

File implementations should determine a file's storage media type, its representation in secondary storage, the code for basic access operations (read and write), and the code for failure recovery (fault-tolerance). The trade-offs in these areas are the most fundamental trade-offs in file implementation design. The file implementation should also determine file cache management since this is an integral part of failure recovery (see Section 4.1.2). File implementations should not determine the particular device on which files are stored or the placement of files within a device; rather, the block placement module should perform this function for all file implementations. This point will be discussed in the next chapter.

2.4.2 File assigner

In a file system with multiple file implementations, there must be a means of assigning files to implementations. In VFS, a file's implementation is determined by the location of the file's directory entry in the file name tree. The problems with this approach have already been discussed.

An alternative approach would be for the user to assign files to implementations by setting a per-file parameter to name the desired implementation. ("User" in this context means both humans and application programs.) Unfortunately, while this approach separates naming from assignment, solving many problems of VFS, it still exposes low-level mechanisms to the user, with negative consequences. Making the user assign files to file implementations is not user-friendly. It requires that the user know the costs and benefits of each implementation and how to apply this knowledge in selecting implementations for files. Also, exposing file implementations to the user has the problems associated with an abstraction violation. For example, it makes modification difficult. When a new file implementation is added, existing files must be checked manually and, where appropriate, moved to the new implementation. Existing programs must be reconfigured before being able to use the new implementation. Gelb examines in detail the problems of exposing file implementations to the user [Gelb89]; note that these problems also apply to VFS, since it too exposes implementations to users.

Another alternative approach would be to completely isolate the user from the assignment of files to implementations. Instead, the assignments could be made transparently by the system using a heuristic and using measurements such as file size, read/write bias, and random/sequential bias. Unfortunately, while the information the system can measure would be helpful in selecting an implementation, it is not enough. Even ignoring the boot-strap problem of assigning a new file to an implementation, the system can not make assignments by itself. For example, the system can not deduce the level of fault-tolerance required by the user. Users must be able to express their needs to the system.

Our approach combines these two alternatives. As in the first alternative, the user must set per-file parameters to control the assignment of files to implementations [Wilkes91]. However, unlike before, these parameters are abstract, they hide low-level mechanism from the user. As in the second alternative, the system takes an active role in assigning files to implementations, using measurements like before and also using the user's parameter settings. For example, a compiler saving a temporary file would set the file's parameters to "temporary file;" the file system would then select the delayed-write implementation or another implementation meant for temporaries. Through parameters, the user provides the extra information needed by the system to make assignments, yet the parameters do not expose low-level mechanism to the user. This approach was pioneered in IBM's system managed data product [Gelb89]. Chapter 4 discusses the proposed parameters in detail.

2.5 Summary

The file system design sketched in this chapter featured multiple file implementations, allowing it to exploit device parallelism where possible, to utilize idiosyncrasies of different media types, to match the levels of fault-tolerance to users' requirements, and to select file representations based on the peculiarities of individual files. To avoid problems with multiple implementations found in VFS, the design uses file behavior parameters to assign files to implementations, isolating naming from implementation assignment, and hiding low-level mechanism from the user.

The next two chapters examine in more detail two aspects of the proposed file system. Chapter 3 examines allocation of disk space for files. Chapter 4 examines at the file behavior parameters for the file assignment module.

Chapter 3

Block Placement

This chapter proposes a disk-block placement algorithm for the type of file system discussed in the previous chapter. A block is the smallest unit of transfer from physical devices to the file system. The file system breaks files into blocks to be stored on physical devices. The *block placement problem* is to find physical space for these blocks. The block placement problem includes both finding space for new blocks and moving (migrating) old blocks. This chapter concentrates on block placement for magnetic disks, which includes picking spindles, cylinders and tracks for the blocks. This chapter assumes that multiple disks are available and that a single block is to be allocated on contiguous sectors on a single disk and is not bit or byte sliced across disks.

Block placement is a fertile area for optimization because of non-uniformities in disk performance and file access patterns. File system designers can reduce seek distance, and thus improve file system performance, by arranging to have the blocks most likely to be used next closest to the disk heads. For example, the designers at Berkeley reduced seek time of BSD Unix by ensuring that the data in a given file is clustered closely on the disk [McKusick84]. File system designers can improve parallelism through placement as well. The designers of the RASH system increased performance for large, sequentially accessed files by placing the blocks of files on different disks [Henderson89].

However, placement optimization is one of many system-wide goals that make up a complicated design equation. Foremost among other considerations is software complexity: as a rule, the more aggressive the optimization, the more complex the software required. The system designer must find an acceptable balance between disk performance and overall system complexity. As file systems move from one to multiple file implementations, software engineering factors force us to reconsider how placement optimization is done.

In the BSD 4.3 fast file system, the abstraction boundary between the file implementation and the block placement algorithm is weak. In essence, the BSD block placement algorithm is part of the file implementation. The BSD placement algorithm uses knowledge of the on-disk representations of files and directories in order to reduce seek distance and rotational latency. This algorithm has been effective at increasing performance over that of previous versions of the Unix file system [McKusick84], but it has problems in a file system with multiple file implementations. Pushing block placement into file implementations requires that the effort of developing and tuning a placement algorithm is duplicated for each implementation. Furthermore, as the number of file implementations increases, it becomes increasingly difficult to ensure that the placement algorithms of different implementations do not interfere with one another unless implementations do not share devices, which introduces other problems.

In a file system with multiple file implementations, it would be best to have a single block placement algorithm shared by all file implementations. This way, the cost of tuning the algorithm is amortized over all file implementations, and the effort of developing a new implementation is reduced since no block placement algorithm needs to be written for it. However, a block placement algorithm shared by all implementations could not make assumptions about file layout. One might wonder if such a generalized placement algorithm can sufficiently optimize placement. This chapter proposes such an algorithm and argues that sufficient optimization is possible while respecting the abstraction boundary around file implementations.

Section 3.1 describes the traces and simulation model used to test the placement algorithm. Section 3.2 describes the part of the algorithm responsible for finding a disk for a block (inter-disk placement). This part of the algorithm was tested by the trace-driven simulator, the primary evaluation criteria being the balance of disk workloads. Section 3.3 describes the part of the algorithm responsible for finding space within a disk for a block (intra-disk placement). This part was also tested by simulation, the evaluation criteria in this case being seek distance. Section 3.4 extends to the proposed algorithm to give file implementations more control over which disk a block is placed on.

3.1 Evaluation method

The algorithms were tested using a trace-driven simulator of a disk array with six disks. This section describes the traces and disk array model. The measurement tools used to take the traces and find parameters for the model are described next. After that, the disk array model is described. Finally, the benchmark traces used in this chapter are described.

3.1.1 Disk request trace points

Built into the disk device driver of HP-UX are three trace points:

- Enqueue. This trace point is where a new disk request is put on the driver's request queue. Specifically, it is just after the new request is put on the tail of the device's request queue and just before **disksort** is called.

- Physical start. This point is where the disk starts servicing the request. It is after the request is removed from the device's queue and packaged into an I/O message and right before the DMA call for I/O message is made.

- Physical completion. This point is where the disk finishes servicing the request. It is right before **biodone** is called by the disk driver's interrupt routine, which means that all of the driver's portion of the interrupt handling is included, but none of the file system's is.

Each trace point posts a six-component record into a kernel buffer. This record contains a time stamp (accurate to one microsecond), a transfer size, a device number (major and minor), a block number, a cylinder number, and a flag indicating read or write. Posting records incurs an overhead of less than 200 microseconds. Posted records were removed by a background user process and saved in a file. This file is stored on a disk dedicated to tracing in order to keep disk traffic due to tracing from interfering with measurements.

After a tracing period (typical trace periods lasted from a few days to two weeks), the raw trace data was processed into final form by combining the three records posted for each disk request into a single record. This record included the time the request was made, queueing time for the request (physical start time minus enqueue time), the service time for the request (physical completion time minus physical start time), the device, cylinder and block numbers, the transfer size, and a read/write flag. The raw trace data contained some inconsistences; for example, completion records were sometimes missing. These inconsistences affected less than 0.06% of the total requests. Requests affected by an inconsistency were fixed by assuming the missing time interval was ten milliseconds.

3.1.2 Disk array model

The disk array simulator was a simple array of first-order disk simulators. No attempt was made to model interference among disks such as contention for I/O bandwidth at a shared junction. Although real-life experience has proven that such interference can be a bottleneck [Chervenak90], we assumed that these bottlenecks would be removed by hardware designers.

The individual disk simulators included a request queue, an elevator scheduling algorithm (taken from the HP-UX file system), and a model of the disk hardware. This hardware model had analytical models for seek time as a function of seek distance and transfer time as a function of transfer size, and it had a probabilistic model for rotational latency. This model and its parameters were based on extensive measurements of the HP7937 disks. The HP7937 has a formatted capacity of 571 megabytes. It has 1396 cylinders and thirteen heads, and it turns at 3600 RPM. The rest this subsection describes how we modeled this disk.

The HP7937 model is based on traces of read requests with transfer sizes of two, four, eight and thirty-two kilobytes. The graphs in Figure 3-1 show service time plotted against seek distance for transfer sizes of two and eight kilobytes. Service time is the request completion time minus the physical start time as measured by the trace points above. The data for four and thirty-two kilobytes was similar in shape.

The measurements were taken on a dedicated machine with no other disk traffic. For each transfer size, 100 requests of each possible seek distance were measured. (Due to size constraints, Figure 3-1 shows only one quarter of the original data, sampled randomly. The model was fitted against all the original data.) All requests of the same transfer size were measured sequentially; however, the order of seek distances was random. A random pause between requests was inserted to randomize rotational latency. To avoid head switches during a transfer, all samples were started on the first block of a cylinder.

The upward "swoop" of the data (see Figure 3-1) was assumed to be due to seek time. This swoop was fitted to a model that is proportional to the square root of seek distance for short distances and linear in seek distance for longer distances. This model is justified by the following physical argument. For short seek distances, the head accelerates for half of the seek and decelerates for the other half. This acceleration pattern leads to square root time. For longer distances, the disk head accelerates to maximum speed before the seek is over and coasts at a constant speed before decelerating. The acceleration and deceleration periods take constant time and the coasting period is linear in the seek distance, so the overall time is linear in seek distance.

A mixed square root and linear curve was fit to the bottom of the total service time

Figure 3-1: Service time versus seek distance data. The resulting formula is:

$$T(d) = \begin{cases} 3.50 + 0.794\sqrt{d} & \text{if } d < 394 \\ 12.8 + 0.0163d & \text{otherwise} \end{cases}$$

which gives seek time (milliseconds) in terms of seek distance (cylinders).

The vertical displacement of the data among different transfer sizes (e.g., the difference in height between the two graphs in Figure 3-1) was assumed to be due to transfer time. The parameters for transfer time were obtained by taking more measurements: service time was measured holding seek distance constant and varying transfer size from one kilobyte to 128 kilobytes. The data fit a linear model the following parameters:

$$T(s) = 5.76 + 0.00103s$$

This formula gives transfer time (milliseconds) in terms of transfer size (bytes). It corresponds to a transfer rate of 0.971 megabytes per second.

Figure 3-2 overlays the seek time and transfer time components we have so far against the measured data. The data in the figure is of 1/16 of the original data, sampled randomly; this makes the analytical curve clearly visible.

The only aspect of the data that remains to be explained is the vertical "thickness" of the upward swoop. This thickness was assumed to be due to rotational latency. Rotational latency was modeled as a uniform random variable with a mean of eight milliseconds. To check the model, the seek and transfer time as predicted by the above models was subtracted from the data. A histogram of remaining time was taken across all seek distances and transfer sizes. This histogram was uniform ranging from zero to sixteen milliseconds.

The model of the physical disk consists of these three models for seek time, transfer time and rotational latency. Folded into the constants of the seek time model are constants for software and other overheads that could not be isolated with the tracing

Figure 3-2: Fitted seek-service time versus seek distance

tools.

3.1.3 Benchmark traces

The benchmark traces used were week-long traces from Red and Cello, two time-shared machines at HP Labs. Cello was an HP PA-RISC machine used by a research group of ten people. Cello was used mostly for reading mail and "news," for editing, for file archiving and for LAN backup. In addition, small-scale development in C++ and C-shell was done on Cello. Red was an HP PA-RISC machine available for all of HP labs. There were approximately 200 accounts on Red, with approximately 20 users logged in during the day. Red was used mostly for reading "notes," editing, and for a few small applications.

Figure 3-3 shows the aggregate throughput demand for each system. Cello was the busier system even though Red served more people; this is because most processes on Red were idle. Cello's heavy activity in off-hours was a result of the nightly news feed which started at 21:00 and incremental LAN backups that ran from 01:00 to approximately 06:00.

Figure 3-3: Average throughput of benchmark traces

Tics labeled with day name are at 00:00 of that day; unlabeled tics are at noon.

3.2 Inter-disk placement

Chapter 2 noted that UNIX has a problem with balancing the workload across disks in a multidisk system. The UNIX file name tree is divided into large sub-trees called "file systems" or "volumes." Volumes are assigned fixed disk partitions when they are created, and all files in a volume are stored on that partition, so in essence inter-disk placement in UNIX is determined by file name. The amount of access to these volumes is non-uniform, i.e., some volumes are used more than others. To keep a UNIX system balanced, system administrators must carefully tune the dividing lines between volumes and the placement of volumes on disks. As illustrated in Table 2.1, this is difficult to do (among other things, a division into volumes is hard to change once it is made). A more flexible volume system such as the one in AFS may mitigate this problem [Sidebotham86]. However, over the long run, as file servers contain more and more data, the sheer size of the problem will make manual disk balancing impractical.

The file system described in Chapter 2 separates naming from block placement: any block can be placed on any disk regardless of the position of the file in the name tree. We take this one step further and allow different blocks from the same file to be placed on any disk regardless of the locations of the other blocks in the file (with one exception, see Section 3.4). But now the question arises: if block placement is not determined by file name, then how is it determined? The remainder of this section compares three alternatives.

3.2.1 Algorithms for inter-disk placement

The three algorithms are called random, bin-packing, and coloring. The *random* algorithm randomly selects a disk for each block. The *bin-packing* algorithm sorts blocks according to access rates, then greedily selects a disk for blocks in order, i.e., after placing blocks 1 through $i-1$, it places block i on the disk with the least accumulated access rate. The *coloring* algorithm builds an interference graph by putting an edge between all pairs of blocks accessed sequentially. It then heuristically colors the graph with disk numbers, giving priority to edges that appear frequently. The coloring algorithm is meant to approximate disk striping systems that guarantee that sequential blocks from the same file are stored on different disks.

Some points about the implementation of the placement algorithms in the experiments. First, to simplify the experiments, the placement algorithms moved entire cylinders around rather than individual file blocks. Second, the packed and colored placements used knowledge of the future, i.e., the access frequencies they used for placement were measured from the experimental trace period itself. As we shall see, giving these algorithms such an advantage does not alter the end results. Third, the "UNIX" placement is the placement given to blocks by HP-UX, which uses the Berkeley 4.3 file system [McKusick84].

3.2.2 Results

Table 3.1 shows the relative disk access frequency according to the different placement algorithms. The table shows that all three proposed placement methods (random, packed, and colored) are effective at fixing the disk load balancing problem. Although packed seems to be best, the difference is small.

Figures 3-4 and 3-5 provide a finer analysis of the effects of different placements. In particular, they show the cumulative distributions of both queue size and queue time. The graphs indicate that load balancing has a significant effect on queue time. On Cello, for example, random placement increased the number of requests waiting less than 10 ms from approximately 35% in the standard UNIX placement to 60%, and decreased the number of requests waiting more than 100 ms from 20% to 6%.

However, the data is still ambiguous regarding the difference between measurement-based placements and random placement, despite the fact that the measurement-based placements used access statistics from the future. Although packed placement came out ahead, the difference is small, and it is outweighed by other factors. In particular, random

Table 3.1: Access distributions to disks on two systems.

Cello				
Disk no	Unix	Random	Packed	Colored
Disk 0	41%	16%	18%	17%
Disk 1	2%	18%	17%	15%
Disk 2	8%	15%	17%	19%
Disk 3	6%	18%	16%	17%
Disk 4	1%	15%	16%	16%
Disk 5	42%	15%	16%	16%
Variance	116%	9%	5 %	8 %

Red				
Disk no	Unix	Random	Packed	Colored
Disk 0	41%	17%	19%	16%
Disk 1	15%	14%	17%	22%
Disk 2	22%	18%	17%	14%
Disk 3	7%	17%	16%	14%
Disk 4	1%	19%	16%	18%
Disk 5	14%	15%	15%	16%
Variance	83%	11%	8%	18 %

Figure 3-4: Cumulative histogram of queue sizes and times on Cello

Figure 3-5: Cumulative histogram of queue sizes and times on Red placement is trivial to calculate, and it does not have to be recalculated over time. Also, random placement is completely independent of any layout policy. Thus, we conclude that random placement is a good algorithm.

3.3 Intra-disk placement

The random placement algorithm used in the previous section not only randomized placement across disks, but randomized placement within disks as well. UNIX, on

Figure 3-6: Cumulative histogram of seek distance on Cello

the other hand, carefully places blocks within disks in order to minimize seek distance [McKusick84]. As Figure 3-6 shows, randomizing placement within a disk results in high seek distances; some other intra-disk placement is needed.

Seek distance can be reduced using an intra-disk placement algorithm called the "organ-pipe cylinder optimization" introduced by Grossman and Harvey [Grossman73] and recently pursued by Carson and Vongsathorn [Carson90] and Ruemmler [Ruemmler90]. In this algorithm, cylinders (or tracks or blocks) are sorted according to access frequency.

The most heavily used block is placed in the center of the disk. The second and third most heavily used blocks are placed on either side of that. The fourth and fifth most heavily used blocks are placed on either side of those, and so on. The resulting arrangement is called the "organ-pipe" arrangement because the graph of access frequency versus cylinder number resembles a set of organ pipes [Wong83].

Figure 3-7 shows the effect on seek distance of applying the organ-pipe optimization with randomized inter-disk placement. In this experiment, a reorganization was done

Figure 3-7: Seek distance after optimization

once per day based on the statistics from the previous day. The resulting seek distance is slightly better than UNIX placement. These results are consistent with other, more detailed experiments on organ-pipe optimization [Carson90, Vongsathorn90, Ruemmler90].

3.4 Inter-disk placement constraints

The previous section suggests a block placement algorithm that randomly selects spindles for blocks. However, fault-tolerance mechanisms often require that some blocks have failure modes independent from other blocks. For example, the parity stripe scheme described in Section 2.1 requires that all blocks in a stripe have independent failure modes. Randomized disk selection can not guarantee these independence constraints. File implementations must be able to express these constraints to the block placement module.

To satisfy this need, the block allocation routine should not allocate blocks individually, but rather in groups. The routine should take the size of the group and return a list of blocks such that each block in the list is guaranteed to fail independently from the others. The routine can raise an exception if the stripe is too wide for the configuration. Blocks are still placed randomly, but now they are subject to these independence constraints.

3.5 Conclusion

The placement algorithm proposed in this chapter takes a different approach from BSD's. It pushes placement out of file implementations and into a module with strong abstraction boundaries around it. Neither the random disk selection nor the organ-pipe optimization require that the representation of files be exposed. For file implementations that need more control over block placement, viz., file implementations that use redundancy for fault-tolerance, the interface between file implementations and block placement is extended abstractly, i.e., without exposing the representation of files.

The results here and elsewhere suggest that the organ-pipe optimization does at least as well as the layout policies in BSD UNIX, indicating that exposing the file representation is not necessary for intra-disk optimization. Given that the coloring algorithm is a good approximation to systems like RASH that use the file representation for inter-disk placement optimization, the the results here suggest that exposing the file representation is not necessary for inter-disk optimization either. Thus file systems with multiple file implementations can use a block placement algorithm that respects the abstraction boundaries around file implementations and can still get sufficient placement optimization.

Chapter 4

User Specified File Behavior

Section 2.4 observed that in a file system with multiple file implementations, files must be assigned to implementations. The user should not have to do assignments; rather, the system should select implementations itself. However, the file system does not have enough information to make a good selection. Section 2.4 proposed that per-files parameters, set by the user, provide the additional information needed by the system to make good selections. The parameters must satisfy two criteria. First, they must supply the additional information needed by the file system to make good assignments. Second, they must abstract away from the details of file implementations; instead, they must reflect the client's concerns, i.e., reflect application-level concerns. This chapter explores on possible set of such parameters.

Section 4.1 examines basic parameters. These parameters control the performance and level of fault-tolerance of files. An example of such a parameter is one to control the throughput of reads on a file. Section 4.2 describes templates and why they are needed. A template is a set of basic parameter values that can be referenced by name. Templates directly reflect application-level concerns; a typical template might be "temporary file" or "important video data." The final three sections of this chapter describe extensions to the parameter scheme.

4.1 Basic parameters

For the purpose of defining the basic parameters, a file is modeled as a abstract data type with four operations: read, write, flush, and recover. The read and write operations are similar to those in most operating systems. The flush operation forces all data written to the file onto persistent, secondary storage. The recover operation models automatic fault-tolerance; it takes as input the state of the file system after a failure and recovers as much of the old contents of the file as possible.

4.1.1 Normal case behavior

Parameters for normal case behavior control the performance of the read, write, and flush operations. Each operation is controlled by two parameters, one for overhead and another for throughput. Overhead is the amount of time an operation takes for the first byte of data accessed; throughput is the time for every byte accessed thereafter. (For the flush operation, the amount of data "accessed" is the number of bytes written since the previous flush.)

Table 4.1: Parameters for normal case behavior

Parameter	Dimension	Description
O_r, O_w, O_f	Time	Overhead (read, write, and flush)
X_r, X_w, X_f	Time/Byte	Throughput

The parameters for normal case behavior are listed in Table 4.1. According to the table, the expected time for a read of s bytes is $O_r + X_r s$. For now, these parameters will be treated as upper bounds. Of course, in any real system such bounds are difficult to implement; we will return to this problem in Section 4.3.

4.1.2 Fault-tolerance

Parameters for fault-tolerance are based on a two-phase recovery model illustrated in Figure 4-1. It is a model of the steps that most recovery algorithms undergo, but it abstracts away from the details of particular algorithms.

Figure 4-1: Abstraction of failure recovery algorithms

At T_0, failure is detected, the file goes off-line, and recovery is automatically initiated; this is the *off-line* phase of recovery. At T_1 the file becomes available again even though recovery continues; this is the *on-line* phase of recovery. At T_2 all recovery is finished and the file goes back to normal operation. There is a small probability that another failure will occur during recovery; in this case, behavior is undefined.

In the first phase, called the *off-line* phase, the file server detects the failure, takes the file off-line (refuses to service operations for the file) and automatically initiates recovery procedures. When recovery proceeds far enough, the file server enters the second or *on-line* phase of recovery, during which read and write operations are serviced but are slowed by on-going recovery work. Eventually, the file server completes recovery and performance returns to normal. There is a small chance that another failure will occur during recovery; in this case, the result of failure recovery is undefined. It is possible for one of the phases to take no time; for example, a mirrored file need not go off-line at all, and a file with a compressed backup might not be available until recovery is complete.

Parameters for this model fall into two categories: performance and data loss. The performance parameters are simple. They control the amount of time each phase may take; T_{p1} controls phase one, T_{p2} phase two. The data loss parameters are more complicated. They are needed because most file implementations improve performance by caching parts of files in volatile memory. During a CPU failure (client and/or server CPU), part of a file might be lost because it has not been flushed from the cache. In this situation, the state of the file after recovery depends on the cache flushing policy of the file implementation. The data loss parameters control the cache flushing policy in an abstract way, i.e., in a way that does not expose the implementation details of the policies to the user. The data loss parameters control two aspects of flushing.

One aspect of flushing that is parameterized is the amount of data that is vulnerable to a failure. This can be parameterized two ways. One way is based on the age of data: all data written to the file more than T_{dl} seconds before a failure will be flushed. Another way is based on the amount of data: at most B_{dl} bytes of data will be vulnerable at any one time. These approaches can be combined: at most B_{dl} bytes of data will be vulnerable, and no data written more than T_{dl} seconds ago will be. For simplicity, we assume the time based approach (T_{dl}) from here on.

Another aspect of flushing that is parameterized is the order in which data is flushed. This parameter controls the the possible states of a file after recovery. The list below describes four values for this parameter. These descriptions assume that the data written by a write operation is broken into equal sized "pages" that are flushed atomically. Unless otherwise noted, the order of flushing described below applies both to explicit flushes (i.e., when the client calls the flush operation) and implicit flushes (i.e., when the file system automatically evicts pages).

- UNIX: no guarantees. Data between explicit flush operations can be flushed in any order.
- Monotonic: no page from a write operation is flushed until all pages from all previous writes are. Within a write operation, pages can be flushed in any order.

- Atomic monotonic: same as monotonic, except within writes either all pages are flushed or none are.
- Atomic flush: either all unflushed pages are flushed during an explicit flush operation, or no pages are (e.g., when a failure occurs during the flush). Furthermore, no implicit flushing occurs, i.e., no pages written between explicit flush operations will be flushed. (In this case, T_{dl} is ignored.)
- Atomic write: either all pages in a write operation are flushed immediately, or none are.

Table 4.2: Summary of fault-tolerance parameters

Parameter	Dimension	Description
T_{p1}	Time	Phase one recovery
T_{p2}	Time	Phase two recovery
T_{dl}	Time	Time data remains volatile
FO		Order of flushing

The parameters for fault-tolerance are summarized in Table 4.2.

4.1.3 Example

The settings in Table 4.3 might define the level of service needed for a low-volume data acquisition program. The modest throughput requirements will allow most disk-based file implementations. However, the zero phase-one constraint will require mirroring. The low data loss constraint and the atomic monotonic flush policy together suggest that very little caching should be done; however, this should not be a problem given the low throughput requirements.

Table 4.3: Basic parameters for data acquisition

Parameter	Value
X_w	0.5 Mbyte/s
O_w	1 ms
X_r	0.5 Mbyte/s
O_r	500 ms
T_{p1}	0
T_{p2}	0.5 hour
FO	Atomic monotonic
T_{dl}	1 ms

4.2 Templates

Templates are defined in terms of the basic parameters. The settings in Table 4.3, for example, might define the level of service for a template called "data acquisition." We expect that in most instances users will be able to use templates defined by the system administrator, so they will be shielded from the complexities of the underlying basic parameter scheme.

Templates serve two important functions. First, they buffer the user from the complexity of the basic parameters. Second, they are an extra level of indirection that can be used to make modifications easier. If the level of performance for executable files needs to be changed, the change should be a simple as a change to a single template.

While templates serve a useful role, one might wonder if the basic parameters do. After all, the system administrator could map templates directly to file implementations. This is the approach taken by IBM's system managed data product [Gelb89]. This alternative has two problems. First, it defeats the purpose of parameters. Directly mapping templates to implementations would still suffer the problems associated with exposing low-level mechanism. Second, the information given by parameters is not the only information used by the system to select implementations; as discussed earlier, parameters supplement measurements such as file size and access patterns.

Thus, mapping templates to file implementations is not enough; however, this conclusion does not rule out a hybrid scheme. Some templates could be defined in terms of basic parameters and others could map directly to file implementations. Directly mapped templates would be for unusual cases, e.g., for specialized file implementations that defy definition by a generalized parameter scheme.

4.3 Variances

In previous sections, we assumed that parameters were hard bounds. This is an unrealistic assumption. In reality, the parameters will have to give expectations; they will have to say what will happen on average. For many applications, such as word processing and program development, loose averages are fine. But for other applications, such as real-time digital video, loose averages are not enough: the correctness of the application depends on data arriving on time.

In order to let the user specify the tightness of parameters, we can extend our basic parameters by adding a variance parameter to the read, write, and flush operations. The variance controls the tightness of the parameter: the closer the variance is to zero, the closer the parameter is to a hard bound. Allowing variances of zero is possible, but would require static, pre-allocation of transient resources such as cache space and network bandwidth. Obviously, this would greatly lower the utilization of the hardware. (The DASH system contains many ideas for implementing hard bounds in a server; however, the DASH system does not do static pre-allocation, and thus can not guarantee that the required resources will be available on demand [Anderson90].)

4.4 File life cycles

The service level a user needs from a file is seldom static; it changes over time. A user working on a paper, for example, may work on it for a week then put it aside indefinitely.

Later, the user may revise it then put it aside again. When working on the paper, the user would like it to have one level of service, viz., fairly prompt performance. When the paper is put aside, however, the user would not care if the level of service was lower. To allow users to specify behavior that changes like this, we extend the parameter scheme to include a *file life cycle*.

To specify a file life cycle, the user creates a labeled, directed graph. The nodes of the graph are labeled with templates. (One node is designated the starting node.) The arcs are labeled with conditions under which transitions occur, e.g., "file has not been opened in three days." Higher-level templates can refer to a file life cycle rather than a simple set of basic parameters. However, the recursion stops at one level: the templates used for the nodes of life cycle graphs must refer to basic parameters, not file life cycles.

4.5 Quotas

An obvious question at this point is "What will keep users from asking for top of the line service for all their files?" To solve this problem, we add quotas to the file system, like disk-space quotas on traditional file servers.

For file servers with multiple device types, a quota system that limits a user to a fixed amount of storage space will not work. Users must be able to use lots of cheap, slow media while being limited in the amount of expensive, fast media they use. This might be accomplished by giving each user a separate quota for each media type; for example, a user might have the right to use several hundreds of megabytes on digital-audio tape but be limited to a few tens of megabytes of disk space. However, this approach also has a problem. It requires too much work from the administrator, because it multiplies quotas by the number of media types. It burdens the administrator further by requiring that each user be given a new quota when new media types are added.

An alternative approach is to invent an "currency" inside the file server and rank the relative "cost" of space on each device type according to this single standard of measure.

For example, a kilobyte of space on magnetic disks might cost ten currency units, while a kilobyte on magneto-optical disks might cost only five. As with traditional quota systems, this approach requires the administrator to maintain only one quota per user. Further, when new media are added, the administrator simply has to assign its space a price.

The pricing system has the additional advantage of smooth scaling. As a server grows in capacity, the administrator can lower the “price” of storage rather than raising the budget of each user. Since a typical server will have many more users than media types, this reduces the administrator’s workload tremendously.

Chapter 5

Summary and Conclusion

This thesis proposed ideas for file systems for future file servers. The central idea was to build the file system with multiple file implementations. This led to another important idea: separate file naming from the assignment of files to implementations and from the placement of files onto physical storage devices. For file assignment, we proposed isolating the user from file implementations with behavior parameters. These parameters allow the users to control the assignments of files to implementations without explicitly naming implementations. The parameter scheme also includes a concept of life-cycles that allow the user to control archiving. For file placement, we proposed a placement algorithm with a strong abstraction boundary between file implementations and the placement algorithm.

The proposed file system promises to increase resource utilization and overall performance:

- Multiple implementations allow the file system to match a file to an implementation suited to the idiosyncrasies of the file and its use patterns. Multiple implementations also allow the file system to optimize for the peculiarities of different media.

- Disk selection that is unconstrained by the name-tree allows better balance of disk workloads.

- The parameters for controlling the assignment of files to implementations ensure that files get the level of service needed by the user and thus use only the resources actually needed. Automatic archiving further ensures that files use only the resources actually needed.

The system also promises to be more user friendly than previous systems:

- The separation of naming from file assignment and placement allows the user to use naming exclusively for organizing files relative to each other. In addition, the separation isolates the user from implementation issues such as the physical system configuration and the characteristics of different file implementations. This, in turn, makes it possible to transparently update software and the hardware configuration.
- Automatic fault-tolerance increases the reliability and availability of the user's data (where needed).

Easier administration:

- Automatic fault-tolerance and automatic archiving relieve the administrator of two burdensome chores.
- The randomized block placement and automatic file assignment algorithms allow automatic system configuration. After the administrator physically installs a new device or new file implementation, the system can automatically use it in an effective manner.

Most of the ideas presented in this thesis have not been tested in an implementation and will doubtless undergo change when going from the drawing board to a real system. However, they should be useful to those looking to build the next generation of large file servers.

Bibliography

[Anderson90] David P. Anderson, Shin-Yuan Tzou, Robert Wahbe, Ramesh Govindan, and Martin Andrews. Support for continuous media in the DASH system. *Proceedings of 10th International Conference on Distributed Computing Systems* (Paris), pages 54–61. IEEE, May 1990.

[Carson90] Scott D. Carson. Experimental performance evaluation of the Berkeley file system. Technical report UMIACS–TR–90–5 and CS–TR–2387. Institute for Advanced Computer Studies and Department of Computer Science, University of Maryland, January 1990.

[Cate90] Vince Cate. Two levels of filesystem hierarchy on one disk. Technical report CMU–CS–90–129. Carnegie-Mellon University, Pittsburgh, PA, May 1990.

[Chervenak90] Ann L. Chervenak. Performance measurements of the first RAID prototype. UCB/CSD 90/574. University of California at Berkeley, May 1990.

[Finlayson87] Ross S. Finlayson and David R. Cheriton. Log files: an extended file service exploiting write-once storage. *Proceedings of 11th ACM Symposium on Operating Systems Principles* (Austin, Texas). Published as *Operating Systems Review*, **21**(5):139–48, November 1987.

[Gelb89] J. P. Gelb. System managed storage. *IBM Systems Journal*, **28**(1):77–103, 1989.

[Gray90] Jim Gray. A census of Tandem system availability between 1985 and 1990. Technical Report 90.1. Tandem Computers Incorporated, September 1990.

[Grossman73] David D. Grossman and Harvey F. Silverman. Placement of records on a secondary storage device to minimize access time. *JACM*, **20**(3):429–38, July 1973.

[Henderson89] Robert L. Henderson and Alan Poston. MSS-II and RASH: a mainframe Unix based mass storage system with a rapid access storage hierarchy file management system. *USENIX Winter 1989 Conference* (San Diego, California, January 1990), pages 65–84. USENIX, January 1989.

[Karels86] Michael J. Karels and Marshall Kirk McKusick. Toward a compatible filesystem interface. *European UNIX Systems User Group Autumn'86* (Manchester, England, 22−24 September 1986), pages 481–96. EUUG Secretariat, Owles Hall, Buntingford, Herts SG9 9PL, September 1986.

[Kim85] M. Y. Kim. Parallel operation of magnetic disk storage devices: synchronized disk interleaving. *Proceedings of 4th International Workshop on Database Machines* (Grand Bahama Island), pages 300–30. Springer-Verlag, New York, March 1985.

[Koehler87] Matt Koehler. GFS revisited or how I lived with four different local file systems. *Proceedings of of the Summer 1987 USENIX Conference* (Phoenix, June 1987), pages 291–305. USENIX Association, Berkeley, June 1987.

[Livny87] Miron Livny, Setrag Khoshafian, and Haran Boral. Multi-disk management algorithms. *Proceedings of SIGMETRICS. '87*, pages 69–77, 1987.

[McKusick84] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, **2**(3):181–97, August 1984.

[Ng89] Spencer Ng. Some design issues of disk arrays. *Proceedings of COMPCON Spring '89*, pages 137–42. IEEE, 1989.

[Ohta90] Masataka Ohta and Hiroshi Tezuka. A fast /tmp file system by delay mount option. *1990 Summer USENIX Technical Conference* (Anaheim, California, June 1990), pages 145–50. USENIX, June 1990.

[Ousterhout85] John K. Ousterhout, Hervé Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A trace-driven analysis of the UNIX 4.2 BSD file system. *Proceedings of 10th ACM Symposium on Operating Systems Principles* (Orcas Island, Washington). Published as *Operating Systems Review*, **19**(5):15–24, December 1985.

[Ousterhout89] John Ousterhout and Fred Douglis. Beating the I/O bottleneck: a case for log-structured file systems. *Operating Systems Review*, **23**(1):11–27, January 1989.

[Parish90] Tom Parish. Volume holographic storage devices, or, storing data in crystals with light. Technical report ACT–BOB–296–90. Microelectronics and Computer Technology Corporation, September 1990.

[Patterson88] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). *Proceedings of SIGMOD.* (Chicago, Illinois), 1–3 June 1988.

[Rodriguez86] R. Rodriguez, M. Koehler, and R. Hyde. The generic file system. *1986 Summer USENIX Technical Conference* (Atlanta, GA, June 1986), pages 260–9. USENIX, June 1986.

[Rosenblum90] Mendel Rosenblum. The LFS file system. Sprite group, Computer Science Div., Department of Electrical Engineering and Computer Science, University of California at Berkeley, 1990. Slides for a presentation.

[Ruemmler90] Chris Ruemmler. Shuffleboard — methods for adaptive data reorganization. Technical Report HPL–CSP–90–41. Concurrent Systems Project, Hewlett-Packard Laboratories, 24 August 1990.

[Satyanarayanan81] M. Satyanarayanan. A study of file sizes and functional lifetimes. *Proceedings of 8th ACM Symposium on Operating Systems Principles* (Asilomar, Ca). Published as *Operating Systems Review*, **15**(5):96–108, December 1981.

[Schulze89] Martin Schulze, Garth Gibson, Randy Katz, and David Patterson. How reliable is a RAID? *Spring COMPCON'89* (San Francisco), pages 118–23. IEEE, March 1989.

[Sidebotham86] Bob Sidebotham. VOLUMES – the Andrew file system data structuring primitive. *European UNIX Systems User Group Autumn'86* (Manchester, England, 22−24 September 1986), pages 473–80. EUUG Secretariat, Owles Hall, Buntingford, Herts SG9 9PL, September 1986.

[Tucker88] Lewis W. Tucker and George G. Robertson. Architecture and applications of the Connection Machine. *Computer*, **21**(8):26–38, August 1988.

[Vongsathorn90] Paul Vongsathorn and Scott D. Carson. A system for adaptive disk rearrangement. *Software—Practice and Experience*, **20**(3):225–42, March 1990.

[Wilkes89a] John Wilkes. DataMesh — scope and objectives: a commentary. Technical Report HPL–DSD–89–44. Distributed Systems Department, Hewlett-Packard Laboratories, 18 July 1989.

[Wilkes89b] John Wilkes. DataMesh ™ — scope and objectives. Technical Report HPL–DSD–89–37rev1. Distributed Systems Department, Hewlett-Packard Laboratories, 19 July 1989.

[Wilkes90] John Wilkes. DataMesh — project definition document. Technical Report HPL–CSP–90–1. Concurrent Systems Project, Hewlett-Packard Laboratories, 18 January 1990.

[Wilkes91] John Wilkes and Raymie Stata. Specifying data availability in multi-device file systems. *Position paper for 4th ACM-SIGOPS European Workshop* (Bologna, 3–5 September 1990). Published as *Operating Systems Review*, **25**(1):56–9, January 1991.

[Wong83] C. K. Wong. *Algorithmic studies in mass storage systems.* Computer Science Press, 11 Taft Court, Rockville, MD 20850, 1983.