A Co-locating Fast File System for UNIX

by

Constantine Sapuntzakis

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

Masters of Engineering in Computer Science and Electrical Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
May 1998

Author .....
Department of Electrical Engineering and Computer Science
Feb 4, 1998

Certified by.....
.....
M. Frans Kaashoek
Associate Professor
Thesis Supervisor

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

A Co-locating Fast File System for UNIX

by

Constantine Sapuntzakis

Submitted to the Department of Electrical Engineering and Computer Science
on Feb 4, 1998, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Computer Science and Electrical Engineering

Abstract

The Co-locating Fast File System (C-FFS) was first developed by Greg Ganger and Frans Kaashoek[1]. Like their earlier design, the Co-locating Fast File System described in this thesis improves small file performance through the use of embedded indoes, co-location of related small files on disk, and aggressive prefetching. This thesis moves beyond the earlier work to present a design and working implementation of a UNIX C-FFS. Unlike the earlier design, this file system provides strict UNIX semantics. The thesis goes into detail about the changes necessary to integrate the co-location and pre-fetching algorithms into a modern UNIX operating system. Novel co-location algorithms, which allow for co-location based on arbitrary criteria, are presented. The thesis presents benchmarks which show that C-FFS achieve near 90% of the disk bandwidth of large file reads on small file reads. Embedding inodes in directories provides most of the gain in bandwidth by allowing the file system to place all data relevant to a file on the same track.

Thesis Supervisor: M. Frans Kaashoek
Title: Associate Professor

Acknowledgments

To my family.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The hard disk-based file system is a core component of today's computer systems. As the principal means of sharing and storing data, the file system needs to be both reliable and fast. Unfortunately, improvements in file system performance have lagged improvements in processor performance and disk bandwidth[6]. The performance gap is most pronounced when applications manipulate small files. This gap is especially felt in the UNIX world, where small tools that manipulate small files are the norm.

This thesis presents a solution: a Co-locating Fast File System for UNIX. The thesis builds on the earlier design and implementation of a Co-locating Fast File System by Ganger and Kaashoek [1]. That implementation was done under a novel operating system called an exokernel[2]. This thesis goes beyond the initial work and design to present a UNIX version of the file system as well as new co-location algorithms. To avoid confusion I will use Exo C-FFS to refer to the original work and UNIX C-FFS to refer to the new design. Unqualified C-FFS is used in statements that are true for both file systems.

Like Exo C-FFS, UNIX C-FFS attempts to reduce the number of expensive disk seek operations on small file accesses. It does this by grouping, or *co-locating*, the information of related small files and their directories contiguously on disk. This enables the file system to read a group of related small files in one large contiguous read. Both file systems aggressively *pre-fetch* data, in that they will retrieve a group of files when one file from the group is requested. This is done with the hope that other

files in the group will be accessed soon and that later disk accesses can be avoided. This technique carries the risk of retrieving useless data from disk and evicting useful data from memory.

One of the goals of the thesis was to make UNIX C-FFS a drop-in replacement for the current file systems under UNIX so as to be able to run real UNIX applications for benchmarking. As such, UNIX C-FFS goes beyond Exo C-FFS in supporting strict UNIX semantics. In addition, UNIX C-FFS generalizes Exo C-FFS's co-location algorithms to support co-location based on a variety of criteria, including file owner, access time, or process id. In contrast, Exo C-FFS only co-locates files from the same directory.

1.1 Background: UNIX File Systems

This section presents some basic background and introduces terminology which will be used throughout the thesis.

A UNIX file system is a collection of files. A file system resides on a *partition*, which is a contiguous range of addressable memory on a random-access storage device.

A UNIX *file* is a sequence of bytes. The file system makes no attempts to generate or interpret the contents of a file - it just stores the bytes passed by the client. Each file has an associated descriptor, called an *inode*. The *inode* includes information on the file size, file times, ownership, permissions, and pointers to the storage associated with the file contents.

A UNIX file system provides a hierarchy of file names via directories. Directories map names to inodes, which represent files or other directories. A well known directory, called the *root directory*, anchors the hierarchy. A directory's contents are not directly interpreted or manipulated by an application. Instead, the file system presents interfaces for adding, changing, and removing names. Multiple names in a given file system can refer to the same file. Storage for a file is automatically freed when the last reference, or *link*, to the file is removed.

Figure 1-1 summarizes the relationship between the various file system objects.

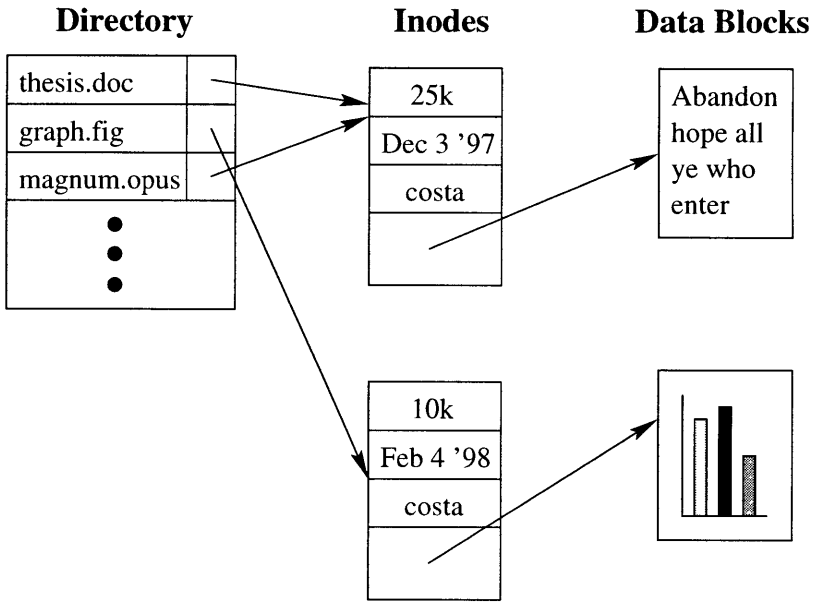

Figure 1-1: Relationship between various file system entities

1.2 The Small File Problem

Current file systems are poorly equipped to handle small files. Reading 100 3K files takes twice as long as read one 300K file. The performance of applications that use small files extensively, such as web servers, compilers, and searches, is adversely affected.

The reason for this performance difference is straightforward. Recall than in a UNIX file system, there are two logical indirections between a file name and the file data. The name points to an inode and the inode points to the file data blocks. In most current file systems, these two logical indirections become two physical indirection. That is, the inode, directory entry, and file data are all stored in separate locations on disk.

Most file systems repeat these levels of indirection for each file access. In the case of the one 300K file, the file system does two indirections and then reads 300K of file data. However, in the case of to the 100 3K files, the file system does 300 indirections to read the 300k of file data. While the file system goes through its many indirections, the application sits idle.

Cycle time of a modern Alpha Processor	2 ns
Time to access random byte of main memory	60 ns
Time to read one byte from disk (no seeks)	40 ns
Average time to seek and start disk transfer	8.5 ms
User noticeable delay	100 ms

Table 1.1: Comparison of various latencies in a computer system

The cost of a physical indirection, or a seek, on disk is expensive, especially as compared processor cycle times. Table 1.1 summarizes some of the latencies in a computer system. On a modern Digitial Alpha processor, a process is forced to wait for almost 1 million instructions before a disk seek is finished.

The disk is not excessively "slow" when it comes to transferring contiguous chunks of data. Modern disks have bandwidths approaching 25 megabytes per second. The 117 us required to transfer 3 kilobytes is dwarfed by the 8.5 ms it takes to do the the average seek. In addition, the transfer time is based on transferring from a single disk. Using N disks in parallel cuts the transfer time by a factor of N.

If indirections are so expensive, then it is necessary to explain why there is only a factor of two between the 100 small files and 1 large file. This is due to caching being done by the disk and by the file system. The file system caches file system blocks in semiconductor RAM, whose access time is several orders faster than disk. The directory block read when manipulating the first small file will probably yield information about other names soon to be referenced. In addition, each inode block contains several inodes, so reading one of the inodes from disk effectively gives the others for free. Finally, the hard disk has a track cache, which store a range of contiguous data on disk. As long the blocks requested from disk are localized, the disk's track cache will absorb many of the reads.

It would seem unreasonable to ask a file system to give equally good performance on reading 100 random files off of disk and it would for one large file. However, in reality, applications do not exhibit random access patterns. A web server will need to fetch images associated with an HTML document. A compiler will compile all the

files in a single directory.

Current file systems do not exploit access patterns to lay out related files contiguously on disk. Nor will file systems attempt to read or write the data of more than one file per disk request. Contiguous layout of relevant data is important when doing pre-fetching – otherwise the file system is either limited in its ability to pre-fetch or the data pre-fetched is not relevant and must be discarded. C-FFS, on the other hand, uses both contiguous layout and pre-fetching to reduce the number of disk requests and potential seeks.

File systems which do not explicitly group are subject to aging difficulties. Since file allocation decisions are done on an individual basis, files will tend to get placed wherevver there is room. The net result is that file data become scattered around disk and the effectiveness of the disk's cache goes down. A grouping file system can help to stave off this difficulty by making the allocation decisions less arbitrary.

1.3 Solutions

There are a couple different algorithmic approaches to co-locating files. C-FFS uses the simplest approach, hard-coded heuristics, but it is worthwile to examine the other potential approaches.

The first approach is the one used today by performance-aware applications. Performance-driven application programmers restructure their programs to use larger files, which are often collections of smaller objects. They are able to explicitly pre-fetch groups of small objects by issuing a large read on the file. This lends excellent performance to this approach. However, applications must be rewritten to take advantage of the larger files. If the objects are irregular, applications can be forced to replicate the allocation algorithms of file systems and in doing so add significant complexity to their code. Another potential disadvantage of this approach is that the names and contents of the small objects are now hidden from easy manipulation by utilities. Finally, this approach needs little support from the file system, so is uninteresting from a file system designer's standpoint.

It is also possible to modify the file system interface to express groups of files. In the second approach, the file system is told which files the application is interested in reading, writing, or allocating as a group. This approach has the advantage of passing information about grouping between the file system and the application. However, it has the large disadvantage of requiring rewrites of every application which wishes to take advantage of this function.

In the third approach, the file system tracks the file access patterns of applications. It then lays out and pre-fetches files based on those patterns. This approach has the advantage of not requiring changes to the file system interfaces or current application programs. Unfortunately, the code for detecting patterns has the potential to be memory and computationally intensive.

The final approach is to use hard-coded heuristics to identify groups. For examples, files in the same directory are often accessed together, and as such, should be grouped on disk. The principal advantage of this approach is simplicity. Of course, such a technique by itself will not be able to take advantage of access patterns which do not fit into one of its heuristics.

C-FFS uses simplest approach, hard-coded heuristics, to determine and maintain groups on disk. In Exo CFFS, the co-location is done based on locality in the namespace. That is, files in the same directory get placed next to each other on disk. UNIX C-FFS retains this approach in its reportoire.

1.4 Contributions

While the UNIX C-FFS design is based on the Exo C-FFS design, the implementation was done from a completely independent code base. Major changes had to be made to the original Exo C-FFS to support UNIX semantics. This thesis describes in detail the UNIX C-FFS design, pointing out differences from the original Exo C-FFS wherever relevant.

UNIX C-FFS is implemented on top of BSD UNIX, which allows for benchmarking against a mature, commercial grade, and reliable file system - namely the Berkeley

Fast File System. The Berkeley FFS implementation was used as the starting point for the UNIX C-FFS implementation, which renders the comparison even fairer, since many of the mechanisms are shared between the two file systems.

Significant changes were necessary in OpenBSD to support a C-FFS. The changes are not unique to OpenBSD, so relevant implementation details are included for the aspiring C-FFS developer.

This thesis also contributes a new co-location algorithm. Key-based co-location generalizes the directory-based co-location presented in Ganger's work by allowing the file system to group on an arbitrary key. If the key chosen is the inode of the parent directory, then the scheme should operate in a similar fashion to the original directory-based co-location algorithm. However, other keys, such as the user ID of the file owner, can be used for grouping.

Finally, the thesis measures the performance of C-FFS on both application and micro benchmarks. Some initial insights are gained into the interaction of the various design elements in the performance of the final system.

1.5 Related Work

The Co-locating Fast File System in [1] evolved from the Fast File System design introduced in [3] which in turn is derived from Ken Thompson's original Unix File System [11].

Achieving a transfer rate equal to the disk bandwidth is a solved problem for large files. [4] describes a method of delivering the disk bandwidth on large files in FFS. The scheme works by detecting sequential acceses to a file and then prefetching blocks ahead of the current read pointer.

Several file systems have properties that improve small-file performance over the vanilla FFS. Silicon Graphics XFS [10] delays allocations of blocks until the last possible moment (be fore the file data is written to disk). As such, it maintains short-lived small files entirely in main memory, vastly increasing their performance. The log structured file system [9] significantly improves small-file write perform ance

by batching all file-system updates into a large sequential (512k) write to an on-disk log. LFS, however, does not attempt to provide high read throughput from disk on small files. Instead, it relies on the main memory cache to absorb the cost of reads. Since LFS lays out files by update time, files that are updat ed together are placed together on disk, possibly yielding beneficial properties for pre-fetching.

Microsoft Windows 98 [5] has an off-line disk optimizer that groups application data and files along with the applications on the hard disk. Loading large applications like Microsoft Word has been sped up by a factor of two by this technique.

Explicit grouping and aggressive prefetching of small files were first demonstrated in the Co-locating Fast File System [1]. This thesis builds upon this work, studying in detail the grouping algorithms.

[8] presents and analyzes policies for prefetching and caching within a theoretical context in addition to running them against traces. They argue for certain desirable properties of a combined strategy that enables their algorithm to operate within a factor of two of optimal. However, they do not present any algorithms for guessing future reference patterns. [7] present a scheme that uses explicit application hints to do informed prefetching and caching. In their approach, the operating system arbitrates amongst hints coming from multiple applications, trying to achieve a global optimum. My thesis focus is complementary to their approach. The file system could be one of clients of this mechanism.

1.6 Summary

The rest of the thesis is organized as follows.

Chapter 2 discusses the origins and the design of the Co-locating Fast Fast File System. It also discusses the co-location algorithms used to improve small file performance.

Chapter 3 examines some of the more interesting implementation details.

Chapter 4 describes the experimental apparatus, experiments, and experimental method and presents the results of the experiments.

Chapter 5 presents areas for future work.

Chapter 6 concludes the thesis.

Chapter 2

Design

The design of UNIX C-FFS is heavily based on the original Exo C-FFS and the Berkeley Fast File System. However, there are some novel elements in the new UNIX C-FFS. Two new data structures, the inode locator table and the external inode table, are used to provide UNIX semantics. In addition, new key-based co-location algorithms are described in Section 2.3.2.

2.1 The On-disk Data Structures of C-FFS

C-FFS divides its partition into *file system blocks*, which usually consist of multiple underlying device blocks, or *sectors*. The *file system block* is the fundamental unit of addressing in C-FFS.

2.1.1 Superblock

C-FFS needs to be able to discover the size of the file system, the location of the root directory, and various other random pieces of state given a raw partition. To solve this problem, the first block every C-FFS file system contains a well-known data structure called the superblock. The superblock describes global file system properties, such as the file system block size, the size of the various on-disk structures, the number of free blocks, and whether the file system was properly shut down. One field of the

superblock is reserved for a special constant. This constant is checked by the file system at mount time to ensure that the superblock is in the correct format.

2.1.2 Inodes in C-FFS

The C-FFS inode contains all the information associated with the file except for the name. Figure 2-1 contains a full listing of the inode contents. The type of the file (directory, regular file, character or block device, symbolic link, socket, or pipe) is described in the inode. The inode also lists the blocks which store the contents of the associated file. The first several entries of the list are physically located within the inode. For files which require more entries in the list, *indirect blocks* are allocated. An indirect block is a file system block devoted to listing other file system blocks. A first level indirect block lists file system blocks. A second level indirect block lists first level indirect blocks. A third-level indirect block lists second-level indirect blocks and so on. Up to three levels of indirection are allowed in C-FFS.

The inode contains a generation number that is changed each time the inode is created. This helps applications which name files by their inode number, such as Network File System (NFS) server, track when an inode number has been re-used to described another file. The 64-bit combination of the generation number and the inode number is effectively a unique identifier for a UNIX file across the life-time of a UNIX file system.

Taking advantage of the fact that modern hard drives write sectors atomically, inodes in C-FFS do not span sectors. This ensures that inodes are updated atomically on disk.

2.1.3 Directories

Directories are also described by inodes in C-FFS. The type field of the inode differentiates them from regular files. This allows common file operations to be used when allocating and manipulating directory contents. This significantly simplifies the directory manipulation code.

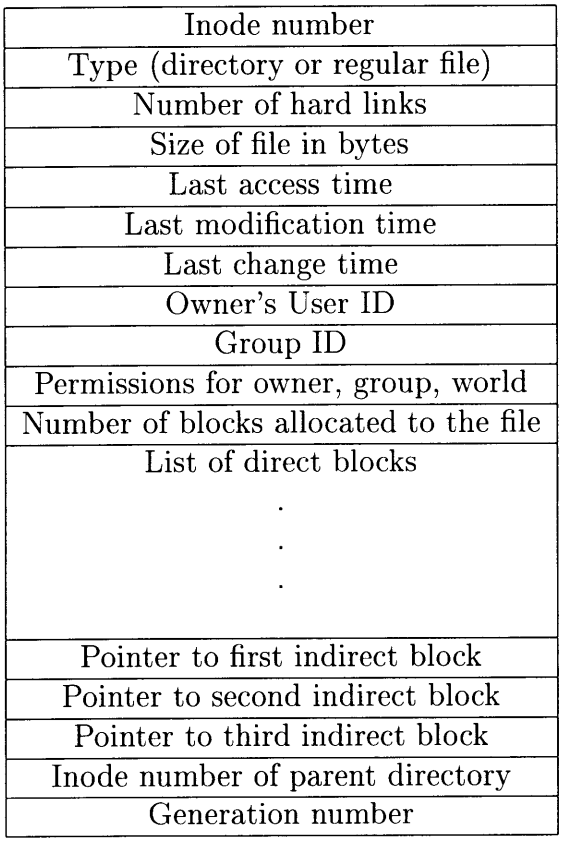

Figure 2-1: Contents of the inode

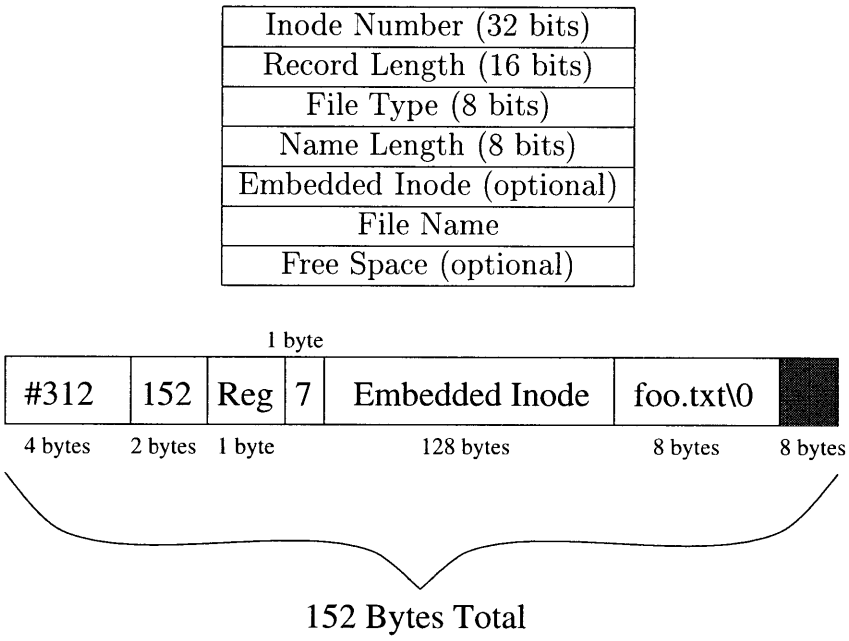

Figure 2-2: Directory entry record format and example

A directory is made up of multiple directory blocks. Each directory block is an independent linked list of variable-length directory entries. Each directory entry contains a name, type, inode number, and usually the inode too. Figure 2-2 summarizes the format.

Unlike FFS, C-FFS does not maintain directory entries for "." and ".." on disk. Both entries are readily faked from the in-core inode, which contains both the inode number of the directory and its parent directory.

The consistency and recoverability of directory information is guaranteed using a couple techniques. First, all new blocks allocated to a directory are written to disk before the size of the directory is update in the inode. Second, as is the case with inodes, directory entries do not span sectors, to ensure atomicity of directory entry updates.

2.1.4 Cylinder groups and free bitmaps

The file system is divided into adjacent extents of blocks called cylinder groups. At the beginning of each group, there is a block with the free bitmap for that group. The bitmap records for each block whether it is allocated or not.

The free bitmap is used to quickly find free blocks or extents. The information is redundant, since the inodes already list the allocated blocks in the file system. As such, the contents of the free bitmaps can be reconstructed from a file system of un-corrupted inodes. This fact is used by the file system to avoid writing the bitmap to disk on every allocation. Instead the bitmaps are written lazily to disk and reconstructed by the file system check utility in case of an unclean shutdown of the file system.

The cylinder group also contains some information on the number of blocks free in the cylinder group.

2.1.5 Inode Locator Table

UNIX requires the file system to be able to retrieve files based on their inode number. In addition, UNIX semantics require us to maintain a constant inode number for the life of a file, even across renames.

The problem becomes how to locate the inode on disk based on the inode number. For other file systems, this is not much of a problem, since they keep their inodes in well known places on disk. However, in C-FFS, inodes are located in directories, which are located at arbitrary locations on disk.

Original Exo C-FFS design

The original Exo C-FFS design used a 32-bit inode number, which was split into a 16-bit directory ID and a 16-bit directory offset. There was an additional data structure, called a directory table which mapped the 16-bit directory ID to the 32-bit inode of the parent directory.

This scheme had the unfortunate effect that the inode number changed whenever

the inode changed directories. Since inodes were relocated on rename, the inode number could change during the file's life-time. This violated strict UNIX semantics.

The static partition of bits between directory ID and directory offset traded the maximum number of directories on a volume or the size of the directories.

New Exo-CFFS approach

In the newer version of Exo-CFFS, the inode is named by its position on disk. This limits the inode to never moving during the life of a file (this is inherently incompatible with our external inode table concept). As a result, directory blocks can not be moved around on disk, which might be desirable if we needed to defragment or group the directory. In addition, when a directory is deleted with active inodes in it, the storage with the active inodes cannot be reclaimed. Instead, those blocks are assigned to a special file off of the root of the file system to be reclaimed when the reference count on the inodes contained therein goes to zero.

UNIX C-FFS approach

The UNIX C-FFS approach is to give each inode a unique identifier. UNIX C-FFS maintains a table that maps each identifier to it the inode's location on disk. The format of the table entry is shown in Figure 2-3. A container inode number of zero indicates that the entry is unsued.

New inode numbers are found by scanning the free entry map at the beginning of the file. The inode locator table uses a slightly more advanced data structure than the free bitmap. The free entry map occupies four kilobytes which is devided among 4096 one-byte entries. Each one byte entry counts the number of used inodes in a 255 inode range. Once C-FFS finds a block of 255 inodes with some inodes free, it must scan the entries individually to find a free one.

The free map describes the status of about 1 million ($4096 * 255$) inodes in a 4k block. To allow for more inodes, another free map can be appended after the initial entries, but before the entries it describes. This process can be repeated to provide for an arbitrary number of inode entries.

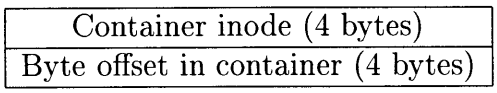

Figure 2-3: Inode locator table entry

Finding an inode on disk by number is a matter of reading the inode from its container at the appropriate offset. If the container's inode information is not in memory, it may have to be read from disk too, thus recursing down the hierarchy to the root of the file system. The recursion stops at the root since the root inode is located in a well-known place on disk. Figure 2-4 presents the pseudo-code for this operation.

The inode locator table is store as a file for convenience. Because of this, it is easy to dynamically grow the file and the number of inodes in the system. This contrasts with the static number of inodes available in FFS. The inode locator table, since it is stored as a file, also has an inode. Its inode is located alongside the root in a well-known place in the first cylinder group.

Even with the extra layer of indirection due to the table, CFFS does not lose the performance benefit of embedding inodes in the directory. First, the UNIX interface does not allow programs to open files based on their unique identifier. Instead, most commands take a file path. As long as C-FFS traverses the directory hierarchy from root to leaf, it should never need to consult the inode locator table, since the inodes for the directories are adjacent to their names. When traveling in the reverse direction (i.e. doing lookups on ..), the inodes for the directories are often cached in-core, so we don't have to try to find the inode on disk, again avoiding a table lookup. In fact, the code to deal with retrieving an inode using the inode locator table was accidentally broken for months and yet the file system could operate for hours of moderate activity (reading e-mail, compiling large programs).

Maintaining the table is not overly burdensome either. Since the contents of the table can be reconstructed from an uncorrupted file system, updates to the table can be written lazily.

```
get_file(inode_number) returns (file)

        if file = is_already_in_memory(inode_number) then
                return (file)


        if inode_number < KNOWN_INODES then
                read inode from known location on disk

                file = init_file(inode)
        else
                container = lookup_container(inode_number)
                offset = lookup_offset(inode_number)
                containerfile = get_file(container)
                inode = read(file, offset, INODE_SIZE);

                file = init_file(inode)

                close(containerfid)
        end

        return (fid)
```

Figure 2-4: Psuedo-code for reading inode

2.1.6 External Inode Table

The external inode table was added to UNIX C-FFS to answer the following important questions: Where does C-FFS place the inode when two or more names in the file system refer to the same inode?

Other possible designs were considered before the data structure was added. For example, the file system could place the inode adjacent to each name. That has the problem of requiring multiple writes to update a single inode.

In Exo C-FFS, the inode stays next to its original entry and all other names point back to the original entry. This approach has the problem of finding a new place for the inode when the original entry and its enclosing directory is deleted and adjusting all the pointers. Exo C-FFS solves this problem by attaching the orphaned directory block to a special hidden directory off of the root of the file system.

UNIX C-FFS takes the alternate approach of moving the inode data to an external table when a second name for a file is created. The directory entries, however, continue to reference the inode through the inode number stored in the directory entry.

Moving the inode out the directory is an expensive process. It makes subsequent accesses across the original name slower, since the inode is no longer contained next to the directory entry. Luckily, hard links are used for only a few esoteric purposes, such as maintaining Internet News spools.

2.2 Ensuring Recoverability of the File System

To ensure recoverability, both FFS and C-FFS order their writes to disk. In the case of FFS, the file system ensures that the link count on the inode is always greater than or equal to the number of names in the file system that refer to an inode. This means that the name of a file is written to disk before the inode is. Similarly, on remove, the inode is cleared before the name is removed.

To maintain ordering, FFS issues an immediate disk write and waits for it to complete. This is known as a synchronous write and is quite slow. However, it provides firm guarantees since once the operation is complete, the data is on disk

and the second write can proceed. In the case of remove, the second write occurs synchronously, to make sure there are never two directory entries with the same name.

FFS uses two synchronous writes on remove and two synchronous writes on create. C-FFS, by placing the inode with the directory entry, can do both operation in just one write, so no ordering is necessary. Thus, C-FFS can do a create in zero synchronous writes and remove in one synchronous writes.

Synchronous writes, since they involve waiting for disk, are slow. Often this wait involves an expensive seek. Given that the average seek is 10ms, this limits the file bandwidth to about 100 files per second.

Finally, embedding inodes impact recoverability. In C-FFS, a corrupted sector can irreperably detach a whole directory hierarchy by wiping the inode of a directory. Since the inode has been wiped out, there is no way of figuring out which blocks on disk belong to the directory and thus no way to read the inodes contained in the directory. Though C-FFS could mitigate this problem by strictly maintaining its locator table, error correction on modern drives and ready availability of backup media seem better ways to attack the problem.

2.3 Algorithms for co-location

The goal of the co-location algorithms in C-FFS is to make related small files adjacent on disk. The extent to which files are related is determined by the access patterns of the file that use them.

2.3.1 Directory-based co-location algorithm

The initial co-location algorithm is based on the observation that files in the same directory are often accessed consecutively (e.g. compiles, reading mail). The pseudo-code for the algorithm is shown in Figure 2-5.

To support the grouping and I/O algorithms, two fields have been added to each inode, a group start field and group size field. Both the group start and group size are expressed in file system blocks. The file system uses this information in the inode

when reading file blocks off of disk. It checks to see if the file block it's fetching is somewhere in the group described in the inode. If it is, it tries to fetch as many blocks from the group as possible, subject to the following constraints: 1) the blocks fetched must be contiguous and 2) the blocks must not already be resident. The second constraint prevents us from over-writing dirty blocks in the buffer cache.

When allocating, only the first block of a file is explicitly placed in a group. Other blocks are allocated using the conventional file system allocation algorithms.

To find a group to put the file block in, C-FFS traverses the inodes in a directory (including the inode of the directory itself). For each inode it traverses, it consults the free bitmap to see if there is any opportunity to place a block in the group or extend the group. If we cannot find a group with room, we allocate a block using the standard allocation algorithm and start a new group with that block.

Each time the group size is updated, every inode that is both 1) located in the same directory and 2) a member of the group is updated.

The size of a group is limited by the file system to 64 kilobytes. In part, this due to the underlying buffer cache implementation, which does not support reads larger than 64k.

2.3.2 A Keyed Co-location Algorithm

The goal of this algorithm is the ability to group on an arbitrary key. A key in this scheme is a 32-bit integer. Related files share the same integer key. For example, the key could be the parent directory's inode number, the creator's uid, or even the last access time of the file.

There are a couple tricks in the previous scheme that are no longer valid. For example, in the previous algorithm, C-FFS searched the directory for relevant groups. In this scheme, we are seeking to be more general, so the alogirthm needs an alternate method of searching for relevant groups. In the previous scheme, the algorithm updated the grouping information by scanning through the directory and rewriting the information in the affected inode. In this scheme, the inodes need not be in the same directory, so the approach can no longer be used.

```
bool group_alloc(inode, lbn, out block_no_found)

if lbn != 0 then
     return (false);

inodes_to_be_considered = { inode->parent_inode } U
                           { other inodes in the directory };

for each potential_inode in inodes_to_be_considered
        prevgroupsize = potential_inode->group.size

        if has_room(potential_inode->group, block_no_found) then
           inode->group = potential_inode->group;

           if prevgroupsize != potential_inode->group.size then
                for each inode in inodes_to_be_consdered
                        if inode->group.start
                           == potential_inode->group.start then
                                inode->group = potential_inode->group;
           return (true);
   end
alloc_block(inode, lbn, block_no_found);
inode->group.start = block_no_found;
inode->group.size = 1;
return TRUE;

bool has_room(group, out block_no_found)

if empty_block in free_bitmap[group.start .. group.start + group.size - 1]
        block_no_found = empty_block
        return true;

if group.size >= maxgroupsize return false;

if free_bitmap[group.start + group.size] != allocated then
        group.size = group.size + 1;
        block_no_found = group.start + group.size
        return true;

return false;
```

Figure 2-5: Initial co-location algorithm

To find related inodes in the new co-location scheme, C-FFS consults a table of in-core inodes that is indexed by key. The table, when queried, yields inodes associated with that key. The design of this table is critical to the performance of the algorithm, so it is best to spend some time describing it.

Unlike the previous scheme, where groups were explcitly described by the inode, this scheme divides the file system into adjacent 64 kilobyte extents called segments. The read algorithm, when reading a block from a small file, attempts to read in as much of the segment as possible.

The table maps keys to the head of a double-ended queue of inodes related to that key. The table is currently implemented as a hash table for efficient lookups. As the algorithm searches for an inode which points to an empty segment, it moves inodes which refer to full segments to the end of the list. New segments are inserted at the head of the queue.

By dividing the disk into fixed segments, we avoid having to update the inodes with group information when the group changes. A caveat is that we'll be more likely to read irrelevant data from disk with new approach. It is unclear at this time how the two factors will balance out.

When allocating a block for a small file, the algorithm examines the segments of the in-core inodes with the same key. Figure 2-6 gives more details.

Large file allocation still goes through the normal FFS algorithms.

This algorithm should benefit from delayed allocation. In delayed allocation, specific blocks are not allocated for the file data until they need to be written to disk. At that point in time, the file system has a better idea of how large the file is. Thus, delayed allocation improves the chances of placing all of the blocks in a small file into the same segment. In addition, allocation can be done on extents of blocks, instead of a block by block basis, leading to efficiency gains when traversing disparate groups.

2.3.3 Group write algorithms

The group write algorithm is invoked when blocks are written back to the disk. The algorithm takes advantage of this opportunity to write out other dirty blocks in the

```
if (have previous allocation)
try to allocate in the same segment;

inodes = lookup_inodes(key);

for inode in inodes
Check the segment related to the inode's first disk block for
room. If found, allocate in that segment and return

move inode to end of queue
If not found, try to find an empty segment and place block in it.

If still no block found, pick a random segment and place the block.
```

Figure 2-6: Key-ed co-location algorithm

buffer cache. It coalesces data from adjacent dirty disk blocks and the original block into one disk transfer. Because of the limitations of the host operating system, these writes are limited to 64 kilobytes in size. However, it is sensible to limit them in any case, so that we don't write out too much data that is going to be changed or perhaps discarded soon. In addition, the larger writes tie up the disk for longer periods of time, potentially delaying subsequent synchronous reads or writes.

2.3.4 Group maintenance algorithms

Whenever we write file blocks out to disk, we can take the opportunity to write the blocks into a different, more optimal location. This still involves the overhead of modifying the block pointers in the inode and updating the bitmaps. However, if the small file's properties have changed significantly since it was originally allocated, it might be worthwile to move it to a new location. Stability is a concern in this scheme. Depending on the pattern of key changes, the file system might find itself constantly shuttling the same data between groups.

Chapter 3

Implementation

UNIX C-FFS is implemented on top of OpenBSD, a freely available BSD UNIX variant. OpenBSD was chosen because of the ready availability of source code for the entire system, good documentation of its internal structures, and its history as a platform for file system experimentation. It also has a fast, robust file system - the Berkeley Fast File System (FFS). Thus UNIX C-FFS implementation is a severely modified version of the FFS code.

This chapter explores some of the challenging issues related to implementing C-FFS on top of OpenBSD and reports on the current status of the implementation.

3.1 Small file grouping and the buffer cache

Before C-FFS fetches a block from disk, it needs to know whether the block is already in the buffer cache, to avoid both caching multiple inconsistent copies of the same disk block and an expensive disk read. Often, C-FFS knows that the block belongs to a specific file and can ask the buffer cache if the that block of the file is already in memory. However, when pre-fetching a range blocks from disk, C-FFS would prefer not to go through the effort of figuring out which file every block belongs to. For pre-fetching, then, it makes more sense to ask the buffer cache whether a given block from the physical device is already present in memory.

To support group writes across files, we need to find a group of blocks contiguous

on disk to write out. This is most easily done if we can ask the buffer cache whether given blocks from the physical device are dirty and in memory.

At the same time, to support delayed allocation, we'd like to be able to keep blocks in the buffer cache whose disk address we do not know yet. We'd like to name those blocks purely by their offset in the file.

Most modern operating systems, including OpenBSD, index their buffer caches solely by offset in a file. To support a C-FFS, they need to support simulatenously indexing block in their buffer cache by offset on device.

In the final scheme that was settled on, the OpenBSD buffer cache was logically divided into two caches. The physical block cache cached only blocks related to devices, such as a disk. The logical block cache only caches blocks that are associated with a files or directories. The read and write interface to the buffer cache was largely unchanged. Instead, requests against physical devices are automatically routed to the physical block cache and requests against files automatically go to the logical block cache.

The same block can appear in both caches. For example, the second block of a directory could also be the 453rd block on the second disk partition.

Special care must be taken when the file system requests a block from logical block cache. Often, the buffer cache must ensure that the block requested is not already in the physical buffer cache (the result of pre-fetching, perhaps). To do this, the buffer cache calls back up to the file system *bmap* function to ask for the physical name of the block. In some situation, this has the potential to create an infinite loop, so the file system can short-circuit the callback by providing both the logical and physical names when requesting a block from the cache.

A couple functions were added to the buffer cache interface. The *bassignlogicalidentity* and *bassignphysicalidentity* functions place already resident blocks in the logical or physical buffer cache, respectively. The latter function is especially useful in delayed writes, where C-FFS assigns the physical location of the buffer on disk at the last minute.

The astute reader will notice a problem at this point. With delayed allocation, we

are asking for a logical block from the buffer cache that has no physical counterpart. To support this, the buffer cache interprets a disk address of -1 returned from *bmap* as a buffer with an unassigned identity.

The final problem with delayed allocation occurs when we attempt to assign the physical identity. There could already be a block with that physical identity in the buffer cache due to a poor pre-fetch decision. Luckily, it is safe to discard that block. Since we are allocating over it, it must contain out of date information.

3.2 Issues with Concurrency Control and C-FFS

When a file system requests a block from the buffer cache, it receives the block back in a locked state. The block remains locked for the duration of the file system's interactions with that block and is finally unlocked when it is released back to the buffer cache. If a file system requests a block that is currently locked by another process, it will go to sleep, waiting for the block to be freed. The file system does not release any of its other locks while it sleeps so the potential for deadlock exists.

A new *try-lock* style primitive was added to the buffer cache. It attempts to acquire the lock on the buffer but returns failure instead of sleeping. The C-FFS group read and write algorithms use this primitive to improve performance and avoid deadlock with the rest of the system.

Another potential issue for C-FFS implementors results from the need to update directory blocks when updating an inode. A directory can be locked for a variety of reasons, including searching or adding names, and it is natural to ask whether a process should wait for the directory to be unlocked before updating the inode. After all, C-FFS might be re-organizing the directory from under us while we're trying to write the inode.

Luckily, UNIX C-FFS easily avoid this issue since access to the contents of disk blocks is serialized by a lock on each buffer. As long as the inode update or directory entry update was done within the context of a single buffer request, the action should appear atomic to the rest of the system.

Another place where this issue comes up is the co-location algorithms. With the directory-based co-location, C-FFS scans through a directory when allocating blocks for a file in that directory. Then, after the allocation in finished, C-FFS must write back the modified inode information. Whether C-FFS should do this with the directory locked is a difficult question. A little more analysis will help to inform an answer.

The locking discipline forced upon C-FFS by the BSD kernel requires the file system to lock in the direction of root to leaf in the directory hierarchy. This precludes acquiring a lock on a parent directory while a child is held locked.

Again, C-FFS deals with the issue by not locking the inode and using the disk buffer lock to serialize access. This could cause problems when multipe processes are doing simultaneous updates on the grouping information directory. If the one with the lower group size goes second, it can erase the previous grouping information. To get around this, the implementation checks to see if the group described is already larger before updating the inode.

3.3 Status

This section talks about the status of the current C-FFS implementation. It concentrates on discussing which elements of the design are still missing an implementation.

Hard links have not yet been implemented under C-FFS. This has not been a hindrance in running common UNIX utilities and development applications. However, hard links are still necessary for specialized applications, so must be present in a fully operable UNIX implementation.

A file system recovery utility, similar to UNIX *fsck*, is necessary to provide recovery in cases of failure or corruption of the file system. Currently, a skeleton implementation verifies the consistency of the directory hierarchy and the inodes contained therin. However, it does not yet rebuild the inode locator table, which is integral to a working UNIX C-FFS implementation.

Keyed co-location algorithms were not implemented for this thesis, though an

implementation of the table structure for the in-core inodes was completed.

Finally, fragments should be reintroduced to increase bandwidth and reduce wasted space in small files. Given the pre-fetch algorithms in C-FFS, this would require major changes to the buffer cache. The pre-fetch algorithms fetch entire blocks into the physically named buffer cache. If the block turns out to belong to multiple files due to fragments, the corresponding buffer must be split up before any portion of it is placed in the logical buffer cache. In addition, the pre-fetch code needs to be able to figure out whether any portion of a block is resident before pre-fetching the block. The current buffer cache is not equipped with functions to deal with these two scenarios.

Chapter 4

Experiments

4.1 Experimental Apparatus

The CFFS times were collected on a 200Mhz Intel Pentium Pro PC. It contains one NCR 53c815-based SCSI controllers attached to 2 2GB Quantum Atlas hard disks on a 10 megabyte/second SCSI-2 bus. The machine contains 64mb of 60ns EDO RAM. A buffer cache of 17 megabytes was used.

The OpenBSD operating system was used for development and experiments. It is a variant of the BSD family of UNIX-like operating systems. It was primarily chosen due to its ready availabity in our research environment and the presence of an optimized, well tested implementation of the Berkeley FFS. In addition, a version of the log structure file system has been implemented, though it is not currently functional.

The January 18, 1998 version of OpenBSD was used as the base for the version with CFFS. The C development tools, namely GCC version 2.7.2.1, were taken straight from the OpenBSD source tree of that date (/usr/src/gnu/usr.bin/gcc).

For all of these examples, the FFS file system used an 8K block size with 1024 byte fragments. The UNIX C-FFS file system used an 8K block size and does not support fragments.

	Operation	Time	Bandwidth	Reads+Writes
FFS	create/write	10.49	6.10	19+1640
	read	11.15	5.74	1050+18
	overwrite	10.27	6.23	27+1550
C-FFS	create/write	10.23	6.26	19+1550
with grouping	read	10.80	5.93	1053+18
	overwrite	10.15	6.31	27+1650
C-FFS	create/write	10.20	6.27	19+1650
w/o grouping	read	10.82	5.91	1051+18
	overwrite	10.16	6.30	27+1590

Table 4.1: Large file benchmark results

4.2 Experimental Questions

The questions addressed by the experiments were:

1. Do the co-location algorithms in C-FFS improve small file bandwidth?
2. Do the co-location algorithms in C-FFS degrade large-file bandwidth?
3. Do the co-location and pre-fetching algorithms in C-FFS improve application performance?

4.2.1 Large File Bandwidth

To ascertain large file bandwidth, a micro-benchmark consisting of operations on large files was used. The micro-benchmark consisted of the following phases:

1. Create/write 64MB file
2. Read file contents
3. Overwrite file contents

The results of running the benchmark are shown in Table 4.1.

The data shows the performance on large files to be essentially unchanged by the addition of co-location and embedded inodes in C-FFS. Since the code for clustered

reads, writes, and allocations for large files is identical to the code used in FFS, this result is not entirely unexpected. However, it confirms that the co-location algorithms do not adversely impact large file performance.

4.2.2 Small File Bandwidth

A small-file microbenchmark was used to ascertain whether C-FFS improved small file bandwidth. The benchmark has four phases:

1. Create 1000 4k files across 10 subdirectories
2. Read the 1000 files
3. Overwrite the 1000 4k files
4. Remove the 1000 files

The cached blocks for the file system being benchmarked are flushed between each phase. Since the expectation is that the file system will eventually migrate all the data to disk, the flushing allows the inclusion of that overhead into the microbenchmark.

The flushing is accomplished by calling the sync operation on the file system. If any block still remain, they are ejected synchronously. The times in the benchmarks include the time require to synchronize the file system.

The results of the micro-benchmark for FFS and UNIX C-FFS are shown in Table 4.2.

C-FFS beats or matches FFS in every category. The most marked improvements in throughput (600% and 52%) are in file create and remove. This is not surprising, since embedded inodes allow us to remove one synchronous write on remove and both synchronous writes on create. However, by eliminating the layer of indirection between the directory name, throughput on read and writes improved by 31% and 12% respectively.

The addition of grouping in C-FFS significantly decreases the number of disk requests on the small file benchmark. On the overwrite phase, the number of disk

	Operation	Time	# of reads+writes	Files Per Second
FFS	create/write	15.34	41+3052	65.2
	read	2.08	1031+21	480
	overwrite	3.98	1031+1021	251
	remove	13.19	41+2033	76
C-FFS without grouping	create/write	2.48	2+1064	403
	read	1.59	1031+30	629
	overwrite	3.55	1031+1030	282
	remove	8.66	34+1004	115
C-FFS with grouping	create/write	2.55	4+181	392
	read	1.47	151+31	680
	overwrite	3.32	167+135	301
	remove	8.62	34+1006	116

Table 4.2: Small file micro-benchmark results

reads and writes goes down by more than a factor of 6 as compared with non-grouping C-FFS and FFS. However, this decrease is not met with a commensurate increase in file throughput. Throughput on reads and overwrites improved by only 6.7% and 7.5% respectively. Most of those improvements were probably due to the decrease in overhead of initiating disk requests. The failure of an order of magnitude difference in disk requests to show up in the file throughput figures is most likely due to the effectiveness of the disk's track cache in absorbing and coalescing smaller reads and writes.

The track cache is especially effective in this case since the benchmark is being run on an empty file system. With an empty file system, the FFS allocation algorithms co-locate the data on disk much like the C-FFS file system with grouping.

At 680 files per second in the disk read benchmark, grouping C-FFS with an 8 kilobyte block size is attaining 5.4MB/s transfer rate off of disk. In contrast, FFS, which reads only 4 kilobytes per file, is only getting 1.9 MB/s from the disk. The 5.4MB/s figure of C-FFS is 90% of the large file bandwidth of C-FFS. Even non-grouping C-FFS achieve an impressive 5.0 MB/s or 85% of large file bandwidth. Embedded inodes are key. They allow all data relevant to a file to be located on the same disk track, thus allowing the file system to take advantage of the on-disk cache.

Finally, grouping does not improve remove performance at all since it is throttled by the synchronous write it must do to remove the directory entry. The performance on create went down by about 3%. This is no doubt due to the overhead of searching directories for groups and the dirtying of large numbers of inodes in updating the group information. Still, the performance hit is impressively small, given the naivete of the algorithm.

4.2.3 Application Performance

The final benchmark involves three applications run daily in our local development environment. The benchmarks were run over the source tree for the exo-kernel operating system, which contains 5955 files spread over 952 directories. Of the files, 5681 of them are under 32 kilobytes.

The operations done on this tree are as follows:

1. A checkout of the entire source tree from a repository stored on a separate local partition. The repository is stored locally, rather than over the network, to avoid variations in the benchmark due to varied network conditions.

2. A compile of the entire source tree from the Makefile at the top.

3. A search the source files for a string that is not present. This search is done with the compiled files in the directories.

The results of the benchmarks are shown in Table 4.3.

The table reports the total run time of the application, the percentage of time spent doing actual work as percentage of the total time spent by the applicaton and the number of reads and writes. The number of reads and writes in this cases includes other partitions.

The CVS checkout was significantly sped up by the addition of embedded inodes. The CPU utilization of the checkout was doubled by C-FFS and the time was almost halved. The real story here is the write requests. The number of write requests decreased by 23% for C-FFS without grouping and 46% in the case of C-FFS with

	Operation	Time	CPU utilization	Reads+Writes
FFS	CVS checkout	5:22	8.9%	9990+39700
	compile	20:01	96.5%	10000+29430
	search tree	0:46.09	4.2%	6350+680
C-FFS without grouping	CVS checkout	2:57	16.6%	8440+30700
	compile	20:19	95.3%	16200+35400
	search tree	0:33.59	6.8%	8330+840
C-FFS with grouping	CVS checkout	2:50	19.5%	8340+21400
	compile	20:00.75	98.6%	6580+20630
	search tree	0:42.80	5.0%	2520+780

Table 4.3: Application performance results

grouping. Since there are roughly 6000 files and directories in the tree, the drop in requests from 39700 to 30700 between FFS and C-FFS without grouping is probably due to the replacement of 2 synchronous writes/inode with one delayed write/inode. The further drop from 30700 writes to 21400 writes is no doubt due to the group writes. However, consistent with the small file benchmark, the drop in disk requests between grouping and non-grouping CFFS is not accompanied by a significant improvement in performance.

The compile seems to be a mostly processor bound task. As such, improving disk bandwidth will not significantly improve overall performance. However, there is still a significant enough disk component that it can be measured and evaluated.

In the compile operation, FFS beats out C-FFS without grouping in both time and processor utilization. This is due to FFS' allocation algorithms, which do a better job of placing new directories than C-FFS does. FFS places new directories in cylinder groups with a greater than average free number of inodes and files in the same cylinder group as their directories. C-FFS has no notion of used versus unused inodes. Instead, it uses the first cylinder group with the smallest number of directories to create the new directory. Directories, then become closer packed on disk which leaves less room for the large object files created by the compile. The object files scatter on disk when they overflow their cylinder groups which hurts performance.

The increase in write requests over FFS in ungrouped C-FFS is indicative of this phenomenon.

On the compile, C-FFS with grouping improves the CPU utilization considerably when compared with C-FFS without grouping. However, the run time just matches that of FFS reflecting additional overheads. One possible source of this overhead is the co-location algorithms, which scan the entire directory looking for a promising group. Offsetting the overheads of the co-location algorithms, there are 30% fewer writes in C-FFS with grouping, no doubt due to group writes. There are also 34% fewer reads than FFS and 60% fewer reads than C-FFS without grouping, which demonstartes the effectiveness of the group allocation in cutting the number of disk requests.

The search operation sees a 60% jump in CPU utilization in C-FFS without grouping versus C-FFS with grouping. This performance gain is attributed to embedded inodes, which decrease the number of seeks that must be done. However, the number of disk requests is higher than FFS, suggesting that the files or directories may have been fragmented. Still, since the performance of C-FFS without grouping is the best of all of the approaches, the fragmentation did not adversely affect performance. This is possible, for example, in the case where a directory block is not contiguous with other directory blocks, but is still right next to the data which is described by its inodes. In this case, the track cache will cache the related data blocks when it goes to read the directory block.

For unknown reasons, C-FFS performs more poorly with grouping on and than with grouping off. This is especially perplexing since C-FFS with grouping has a lower number of disk requests in both reads and writes and the algorithms were supposedly optimized for this scenario. This is definitely an area that needs more study.

4.3 Conclusions

Embedded inodes are a good idea. They significantly increase small file bandwidth on creates by removing the need for synchronous writes. Embedded inodes also increase

performance across the board by removing a level of physical indirection between the directory and entry. Though they introduce significant complexity into the design and implementation of the file system code, embedded inodes are worthwhile from a performance standpoint.

The small file microbenchmark shows that C-FFS reaches 90% of the large file bandwidth when reading off of disk. In contrast, FFS only attains about 33% of disk bandwidth. The placement of all relevant file data in the same track, which is possible with embedded inodes, is critical to the performance difference.

Disk request counts are essentially useless for predicting the throughput or runtime of both microbenchmarks and applications. They are rendered useless by today's disks, which seem to absorb common sequential or near-sequential access patterns in their cache. Disk access patterns, such as those derived from tracing tools to be discussed in Chapter 5, should be a much better predictor of performance.

The measurements, however, do not support the hypothesis that the co-location algorithms are worth the complexity of retooling the buffer cache and file system allocation algorithms. Though they significantly reduce the number of disk requests necessary, most of the performance gains seen in the microbenchmarks come from the embedded inodes.

However, the measurements done for this thesis were limited. The real test of the effectiveness of the grouping algorithms come as the file system fills and ages. As a file system fills up and as numerous create and remove operations are done against it, files tend to get placed wherever they fit rather than grouped on disk. Earlier studies by Ganger[1] show that files in the same directory in an aged Fast File Systems have poor locality on disk. Similar studies need to be done on an aged UNIX C-FFS to ascertain the effectiveness of grouping algorithms over the long term.

Chapter 5

Future directions

C-FFS, though functional, is still very much a work in progress. Significant work in the implementation, testing, and measurements are necessary before it is a viable replacement for the native UNIX file system.

The actual implementation of the keyed co-location algorithms is an integral part of continuing the work on UNIX C-FFS. The benchmarks results on the initial design and implementation will no doubt cause further iterations of the design. Since the performance improvement due to the initial directory-based co-location algorithm is scant, many iterations need to be done to refine the design.

5.1 Testing

Any production file system must be extensively tested before it is put into use. Since it is the core component for storing and sharing information in most computer systems, when it becomes unavailable, the system loses many if not all of it functions. Restoring the information from backups can be time-consuming.

Though the file system designer and implementor can be careful, not all of the flaws will be caught initially. Some problems will only manifest themselves after hours or days of operation and then only once. Confidence in the stability and reliability of a file system grows with the number of hours of continual use. Variety is also important. Different workloads stress thef file system in different ways and uncover

different bugs. Certainly, since UNIX C-FFS's use has been limited to one person, the author, it does not meet the criteria of a well tested system.

5.2 Future Measurements and Experiments

More data needs to be collected about file and disk access patterns of I/O bound applications. The access pattern data can be used to fashion more effective grouping algorithms than the ones currently present in C-FFS. In addition, more, different application benchmarks and detailed static analysis of disk layout would contribute greatly to the understanding of the file system. This section goes into detail about future experiments and the tools they will be built upon.

To collect data about file and disk access patterns, the OpenBSD kernel was instrumented to record data about the occurrence and duration of file system operations. Though the mechanism for collecting and processing the data is fully implemented and woking, there was insufficient time to fully implement and run the experiments mentioned below for this thesis.

5.2.1 Tracing driver design

To record the duration of an operation, C-FFS calls the trace recording routine at the start and end of the operation. This is accomplished by manually inserting procedure calls in the file system and disk driver code. Figure 5-1 shows an example of an instrumented code fragment.

The trace recording routines note the event that occured (e.g. `VFS_READ`) along with the current time. Since multiple processes can be in the file system simultaneously, the process identifier of the current process is also recorded. In addition to the event, the trace recording routine accepts a payload of bytes which is appended to the trace record. The payload is not interpreted by the trace driver. A full breakdown of the trace record format is shown in Figure 5-2.

The trace events are recorded in a trace buffer located in kernel memory. The trace buffer is exposed read-only through a device which can be mapped into the

```
void
biodone(bp)
        struct buf *bp;
{
        if (ISSET(bp->b_flags, B_DONE))
                panic("biodone already");
        SET(bp->b_flags, B_DONE);               /* note that it's done */

        if (IS_UFS(bp->b_vp))
         record_it(5, ID_BIODONE, bp, bp->b_vp, bp->b_lblkno, bp->b_blkno);
...
}
```

Figure 5-1: An example of the instrumentation

Size of payload in bytes	1 byte
Current Time	8 bytes
Current Process ID	2 bytes
Event ID	4 bytes
Payload	Variable size

Figure 5-2: Format of trace buffer

address space of client applications. The trace buffer is a circular queue. Through the *ioctl* interface, the application can query the trace device for the head and tail of the queue. To delete data from the queue, applications with write priveleges on the device are allowed to set the head of the queue (as long as the new value for the head is valid).

The file system can generate a large quantity of trace events, especially when tracking activity in the buffer cache. To reduce the amount of data that needs to be processes, the trace events are divided into classes which can be selectively turned on and off for different experiments.

5.2.2 Tracing utilities

Several utilities work together to extract and process the data. The input to these utilities is the raw trace buffer and the output of the utilities is a nested graph of the

operations. The nested graph allows programs and humans to easily ascertain which operations were spawned by other operations. This enables various questions such as "How many disk reads occured for as a result of create operations" to be answered by the data.

The first utility is a trace server. The trace server is a TCP server which dumps the contents of the trace buffer to any client that connects to it. The trace client is equally simple. It connects to the trace server and redirects the raw data from it to a file specified by the user.

The two utilities allow the trace data to be written to the hard disk of machine other than the one that is being measured. This prevents distortions such disk writes and flushing of cached file system blocks from skewing the data. Of course, the trace server requires some processor time to run, so it will affect application performance. But it is significantly better than the alternative which contains no server at all.

Several scripts process the binary data from the trace buffer and convert it into more or less human-readable form. Figure 5-3 shows the output of the various stages of the scripts.

The first script changes the binary data to human readable numbers and event identifiers. It does this by parsing the C header files which define the indentifiers.

The second script pairs events. The pairing is done as follows. First, pairing is only done between events from the same process, so each event is first place into a bin based on its process ID. Then, an event whose names end in `_DONE` (e.g. `VFS_READ_DONE`) are paired with the most recent events whose name is the prefix `VFS_READ`. Certain operations (like `BIODONE`) terminate multiple events whose names are not a prefix. These operations are specially cased in the code. If two events are matched, the script outputs a line containing the time of the first and second event along with the process id, name, and payload of the first event. The payload and name of the second event is currently discarded since most of the terminating events do not provide additional information short of the end time of an operation. Finally, if an event cannot be paired at a given point in the data, it is placed in the bin for possible future matches.

The final script takes the paired events and creates the call graph. The call graph

Stage 1: Dump of trace buffer

```
1:3611702067 0 BREAD 4046031840 4035084672 16 8192 0
1:3611702903 0 DISK_READ 4046031840 16
1:3612110447 0 BIODONE 4046031840 4035084672 4294967295 16
1:3612121931 0 BRELSE 4046031840 4035084672 4294967295 16 1057296
1:3612139735 0 BRELSE 4046031704 0 0 0 8208
1:3612140761 0 BREAD 4046031840 4035084672 560 1024 0
1:3612141453 0 DISK_READ 4046031840 560
1:3613143866 0 BIODONE 4046031840 4035084672 4294967295 560
```

Stage 2: Pairing of events

```
1:3611702067 1:3612121931 0 BREAD 4046031840 4035084672 16 8192 0
1:3611702903 1:3612110447 0 DISK_READ 4046031840 16
1:3612140761 1:3613148396 0 BREAD 4046031840 4035084672 560 1024 0
```

Stage 3: Call graph

```
1:3611702067 1:3612121931 0 BREAD 4046031840 4035084672 16 8192 0
        1:3611702903 1:3612110447 0 DISK_READ 4046031840 16
1:3612140761 1:3613148396 0 BREAD 4046031840 4035084672 560 1024 0
```

Figure 5-3: Various stages of output

Operation A spawned operation B if:

$$A_{\text{pid}} = B_{\text{pid}}$$
$$A_{\text{start}} < B_{\text{start}}$$
$$A_{\text{end}} > B_{\text{end}}$$

Operation A spawned an asynchronous operation B if:

$$A_{\text{pid}} = B_{\text{pid}}$$
$$A_{\text{start}} < B_{\text{start}}$$
$$A_{\text{end}} < B_{\text{end}}$$

Figure 5-4: Rules for ordering operations

relation between A and B is defined in figure 5-4. Currently, asynchronous operations are poorly handled. Namely, the call graph program does not attempt to determine which operations were spawned by an asynchronous operation. Instead, it assumes alll operations are spawned by synchronous operations. Handling asynchronous operations effectively would require another event in addition to the start and end time - the point at which the procedure that spawned the asynchronous operation returned to its caller. This would allow the program to differentiate the case where the asynchronous operation spawned an event and the case where the caller called a second operation.

Utilities for static analysis of the allocation patterns of FFS and C-FFS file system are also useful for answering questions about the efficacity of the allocation algorithms.

5.3 Major questions

The following experimental questions were largely unanswered by the thesis but are key to proving the effectiveness of the co-location approach in C-FFS.

1. Do small files matter?

2. Are small file accesses significantly slower than large file accesses?

3. Are small file accesses often grouped?

4. Do small files exhibit poor grouping behavior under current file systems?

5. Are co-location algorithms in C-FFS succesful in grouping small files, even under the stress of aging?

6. Do the grouping algoirthms in C-FFS improve bandwidth on small file accesses?

7. Do the co-location and pre-fetching algorithms in C-FFS result in improved application performance?

8. Do the grouping algorithms in CFFS adversely impact large-file performance?

Chapter 6

Summary

This thesis presents the design and implementation of a Co-locating Fast File System for UNIX. The Co-locating Fast File System concentrates on improving small file performance by reducing the number of indirections the file system requests of the underlying device. Specifically, embedding inodes in directories remove a level of physical indirection from the structure of the file system. In addition, the observation that files accesses are often grouped allows us to remove even more indirections by pre-fetching groups of files. Pre-fetching is most effective, however, if files are layed out contiguously on disk. As such, a special class of layout algorithms called co-location algorithms are key to file system performance.

Two different algorithms for laying out files contiguously on disk are presented. The directory-based co-location algorithm comes directly from Ganger and Kaashoek's earlier work. The key-based co-location algorithm is a novel algorithm presented for the first time in this thesis. It generalizes the earlier directory-based co-location to support grouping on various keys. Suggested possibilites for keys include access time, user id, or even parent directory.

A UNIX file system is required to maintain the logical indirection between a file name and its inode. This indirection allows for multiple names in the same file system to refer to the same file. It also allows the programs to assume a constant file identifier for the lifetime of a file. Removing the level of physical indirection in the presence of a logical indirection adds complexity to the design. A couple alternatives are discussed.

UNIX C-FFS adds two data structures, the external inode table and the inode locator table, to provide constant inode numbers and multiple link support.

The buffer cache of the host operating system, OpenBSD, had to be significantly changed to support the group pre-fetch algorithms. The buffer cache was split into two caches: a physical block cache and a logical block cache. Individual blocks could appear in both or either. The new buffer cache structure allowed the pre-fetch algorithm to read blocks into memory without knowing which files they belonged to.

Performance measurements on the file system confirm the value of embedded inodes. By allowing the file name, inode, and data to reside in the same track, small file bandwidth is improved to the point where it is 90% of large file bandwidth. Measurements also indicate that large file performance does not suffer because of the addition of the new algorithms. Application benchmarks are not as conclusive about the benefits of co-location and embedded inodes.

Very little of the performance improvements seen in the benchmarks comes from grouping. This is mostly due to the excellent performance of all file systems vis-a-vis grouping on empty partitions. In addition, modern disks help by caching ranges of contiguous data. The advantages of grouping are expected to become more evident on a fuller file system in the presence of aging.

Future work will concentrate on developing new co-location algorithms based on the studies of detailed file system access patterns. These patterns will be collected by a tracing mechanism in the kernel which instruments file system operations all the way down to the disk driver. Experimental questions for validating and directing the future work were presented.

Bibliography

[1] Gregory R. Ganger and M. Frans Kaashoek. Embedded inodes and explicit grouping: Exploiting disk bandwidth for small files. In USENIX, editor, *1997 Annual Technical Conference, January 6–10, 1997. Anaheim, CA*, pages 1–17, Berkeley, CA, USA, January 1997. USENIX.

[2] M. Frans Kaashoek et al. Application performance and flexibility on exokernel systems. In *16th ACM Symposium on Operating Systems Principles. Saint-Malo, France*, pages 52–65, October 1997.

[3] M. K. McKusick, 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, August 1984.

[4] L. W. McVoy and S. R. Kleiman. Extent-like performance from a UNIX file system. In USENIX Association, editor, *Proceedings of the Winter 1991 USENIX Conference: January 21-January 25, 1991, Dallas, TX, USA*, pages 33–43, Berkeley, CA, USA, 1991. USENIX.

[5] Microsoft. Windows 98 beta 3 release - feature overview, January 1998.

[6] David Patterson and John Hennessy. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990.

[7] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 79–95, Copper Mountain, CO, December 1995. ACM Press.

[8] Kai Li Pei Cao, Edward W. Felten, Anna R. Karlin. A study of integrated prefetching and caching strategies. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, pages 188–197, New York, NY, USA, May 1995. ACM Press.

[9] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 1–15. Association for Computing Machinery SIGOPS, October 1991.

[10] Adam Sweeney, Don Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS file system. In USENIX Association, editor, *Proceedings of the USENIX 1996 annual technical conference: January 22–26, 1996, San Diego, California, USA*, USENIX Conference Proceedings 1996, pages 1–14, Berkeley, CA, USA, January 1996. USENIX.

[11] K. Thompson. UNIX time-sharing system: UNIX implementation. *Bell Sys. Technical Journal*, 57(6):1931–1946, 1978.