Twiggy Chan, Jeff Klann, James Robertson

6.824 Final Paper
BSFS: B-tree Structured File System
12/06/01

Abstract

The B-Tree Structured File System (BSFS) provides an implementation of a traditional filesystem that is stored in a single, persistent B-Tree backed by a logging and transaction subsystem. It provides a low-complexity file-system with usable performance that complies with most standard POSIX semantics and provides nearly all the functionality of NFS. BSFS also guarantees file system consistency after a crash using transactions and logging, but it hides the complexity of logging in its B-Tree package. This paper explores the advantages and disadvantages of BSFS over a traditional file system, it examines key design and implementation issues, and it details results and performance.

Introduction

The typical UNIX implementation of a file system is awash with complexity. *inodes* are data structures used to stores file attributes (metadata) and the disk addresses of file data. d*irectories* are special files containing a data table of directory entries (inodes and filename pairs). *Allocation tables* and *free lists* are used to track open space on the physical disk and open inode numbers. Large files may have additional structures that store disk addresses when the fixed-size inode cannot hold them all. A single file write requires at least four updates: the inode must be changed to reflect the write (i.e. new file size and new disk addresses), data must be written to the disk addresses, the corresponding directory entry must be modified, the list of free disks blocks must be changed, and the file modification time must be updated in the inode. A single file lookup is similarly complex. Each directory in the hierarchy from root must be loaded and searched in order to load the next lower directory on the hierarchy, until the file's inode number is found. Then the inode must be loaded and an additional disk seek must happen to begin loading data from the file. Thus, unless caching is employed, a single file lookup requires O(n) disk accesses, where 'n' is the number of layers between root and file. This is adequate for small trees of files and directories, but suffers from poor file lookup times for hierarchies that have grown very tall, since another table needs to be read for each subdirectory. Files that are located at the end of long subdirectory chains have much worse lookup times than file that are located near the top of the hierarchy (unless the directory was cached). Also, directories with large numbers of files suffer from long lookup times, because of the linear search of the directory listing. The Unix Fast File System (FFS) and other traditional UNIX filesystems use clever caching algorithms to bring about reasonable performance. Nonetheless, the immense complexity of a UNIX filesystem is problematic for at least two reasons: 1) these highly-complex processes require clever mechanisms to prevent dismal performance (and even then performance isn't stellar), and 2) these highly complex filesystems make it difficult for a moderate computer scientist to understand, modify, improve, and generally do research on filesystem design.

Another difficulty the traditional UNIX filesystem suffers is its lack of *crash-recoverability*. FFS, for example, is only moderately fault-tolerant without additional help. Because a logical write operation requires multiple physical writes, a crash in the middle of a logical write could leave the file system in an unintended state – a file could exist in a directory listing, but not contain any valid data, or worse, it could contain someone else's private data left over on the disk after a delete operation. FFS relies on boot-time utilities to examine the file system's meta-data to repair these integrity problems. At best, this is a time-consuming process (perhaps several hours), and at worst it could be unable to fix a disk inconsistency.

In response to these problems, we developed a B-Tree Structured Filesystem (BSFS). BSFS indexes all filesystem data (including data and metadata) in a single, persistent B-Tree using full-pathname-indices. This turns out to have turns out to solve a number of the *complexity* and *crash-recoverability* issues in a system like NFS while still providing reasonable performance. BSFS can lookup and access any file directly in O(log n) time (where n is the number of files in the system). Not only does this offer high-performance when doing random file lookups across the whole file system, it also decreases the complexity of the traditional hierarchical-lookup. All files are stored directly in the B-Tree, eliminating the need to keep *free lists* and *allocation maps* and even extent-lists in the metadata. Additionally, our B-Tree package (BerkeleyDB from Sleepycat Software) provides *transactional-data-store* functionality, allowing us to design transactional semantics that always leave the file system in a consistent state on startup. This functionality is similar to a journaling filesystem like IBM's JFS; BerkeleyDB logs all changes to the filesystem and commits them only after an update has been totally completed. BSFS is advantageous to JFS from a design perspective, however, because it implements journaling at a much lower complexity – nearly all the log writing and transaction processing are obfuscated by BerkeleyDB; our design only had to worry about where to open and close a transaction.

Overall, BSFS attempts to improve upon a traditional UNIX file system by significantly decreasing *complexity*, adding transactional *fault-tolerance*, and maintaining reasonable *performance* near that of NFS. In the following sections, we will discuss how this was done and what results were obtained,

focusing on the following key design issues: 1) how to order file name keys in the B-Tree in a way that directory listings are fast, 2) how to represent file metadata and data, 3) how to implement rename() and file links, and 4) how to arrange transactions to ensure filesystem consistency while maintaining performance.

Design

BSFS (B-tree Structured File System) uses a single B-Tree structure to organize file system data, while maintaining the hierarchal UNIX semantics. This persistent B-Tree can then be accessed using an NFS loopback server, so it can be treated like a traditional file system. The system consists of three layers: a network file system interface, a BSFS abstraction to connect the B-Tree to the file system interface, and a persistent B-Tree implementation.

The diagram in Figure 1 illustrates the organization of the BSFS system. A network file system interface, sfsusrv, handles requests from clients and interfaces with the BSFS abstraction layer to get file data through POSIX-like file system calls. The BSFS abstraction layer interprets the file system calls into function calls on the underlying B-Tree implemented by Berkeley DB. We will return to the details of this diagram in the implementation section later on.

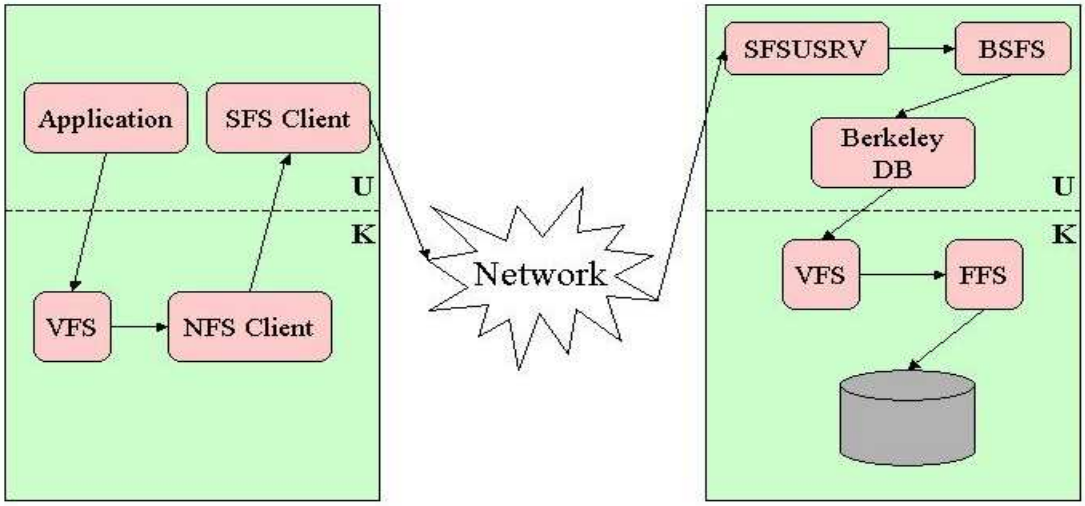

Figure 1 – BSFS components

Representing File Data and Metadata

Files are stored in the B-Tree as arbitrary-length data-blocks keyed on full pathname. Duplicate keys are not allowed. Key-entries exist for every file and directory in the system, beginning with root ("/"). (This is shown in more detail in Figure 2, next section.) When file data is read or written, the data in the B-Tree is modified directly; the only persistent structure in BSFS is the B-Tree. File lookups occur in O(log n) time (where n is the number of total files), thanks to the structure of the B-Tree.

Each file contains two pieces of information. The first record is the file's data. The second, the metadata, contains the information traditionally stored in an inode (except for disk-address listings, which are unnecessary because the B-Tree manages physical storage). Thus some 50 bytes of data are stored per file, including file size, file type, permission bits, user and group ids, file size, and last access, modification, and change times.

This design for file data and metadata achieves a much greater simplicity than traditional file systems, in that most operations can occur in one or two physical disks accesses; there is no need to modify directory entries because they are implicit in the B-Tree, and there are no complex structures to track free inodes and free disk blocks, because this complexity is hidden by the database associated with the B-Tree.

Ordering file name keys

Another elegant simplification BSFS achieves concerns *directories*.

Directories are special files that have an empty data record and a file type of *directory* in the metadata. Only directory metadata is stored in a directory-file; the actual contents of a directory are retrieved on the fly from the B-Tree when it is requested. These directory files are necessary both for metadata and to give the directory an entry in its parent directory, which would be difficult to determine otherwise.

The directory structure itself is stored implicitly in the B-Tree by modifying the B-Tree sort function. The modified sort function provides standard lexicographical key sorting, with one modification: if key A has more '/'s than key B, it is always greater than key B. With this modification, when keys are inserted into the database, all the files in a directory are grouped together. For example, if we insert "/a", "/b/c", and "/x" into the database, then "/a" and "/x" are stored before "/b/c/" even though lexographically that wouldn't be true. Thus we can use the structure of the B-Tree to generate directory listings on the fly, without duplicating this information elsewhere in the file system.

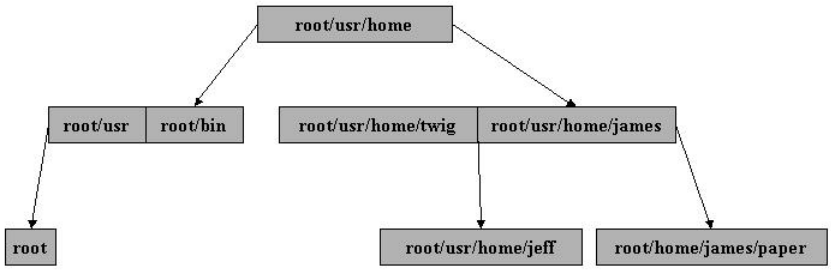

Figure 2 – B-Tree Layout

Links and rename

Hard links can be implemented using database records that represent a link instead of a file. Each hard link record can be keyed using its new filename, which contains the original file's pathname for lookup. Unfortunately, there is no way to re-key a record in BerkeleyDB, apart from copying all of the file's data to a new record. Since this could take too long for a large file, each file would remain keyed under its original pathname. This means the hard links are really glorified, file-system managed symlinks, with a master file containing the file data, and several special hard link files pointing toward the master file. This design is effective, except complexities develop if the user attempts to remove a master file while hard links still point to it. Removing a file to which there are hard-links causes it to still remain but be marked as permanently hidden, eliminating the need for an expensive re-keying operation. However, unlike in FFS or other traditional Unix file systems, a user could not create another file with the old master file's name as long as hard links remain to it, because each database record must have a unique key.

Various schemes can be concocted where modified pathnames are used to key the new file so that it does not conflict with the hidden, linked file, but they all ruin the elegance of the system with their excessive complexity. In the end, hard links can be implemented, but are probably not worth the trouble. Our system does not implement hard links.

Rename, however, is harder to dismiss since it is much more commonly used. In traditional Unix file systems, rename is implemented in a similar manner to hard links, by creating and removing directory entries that point to the same inode. We implemented rename using the somewhat problematic hard-link design above. When a file is first renamed, it is marked as permanently hidden and a hard link is created for it. If it is renamed again, the original link is deleted and a new one is made. It functions well, but, as in the hard-link design, it is impossible to duplicate the original file's name until the renamed file is deleted. The system could be improved with more complex hard-linking, as above, but all possible solutions to this problem are either horribly slow or horribly complex. Therefore, the issue remains in our design, but simplicity is maintained.

Transactions

A key element of BSFS is *fault-tolerance*. A crashed system must be able to restart within minutes and without any inconsistencies in its file data.

This is done using BerkeleyDB's *transactional data store* subsystem, which supports *logged, transactional* writes. Whenever a physical write occurs, it gets logged to a log file. Later, when BerkeleyDB *checkpoints*, it safely executes all committed, logged transactions on the database, and the log can be deleted. Operations inside uncommitted transactions are never checkpointed from the log to the database, so if a fault occurs while inside a logical operation wrapped in a transaction, then the resulting physical operations are never inserted into the database, leaving the system in a consistent state. When the system boots, BerkeleyDB must only read through the log entries that have occurred since the last checkpoint, making startup quick provided checkpoints occur at regular intervals.

Therefore, BSFS can guarantee *atomicity, consistency, isolation* and *durabilty.* This means that a file operation is not committed until all *physical* writes associated with it have been logged, leaving the system in a consistent state when recovering from a crash – either a logical operation has committed or it appears to have never happened at all. Furthermore, BSFS logs both file metadata and file data, whereas most journaling file systems only log file metadata. We chose to do this to make all operations on the file system consistent, although this comes at some cost in performance.

Every update to the file system in BSFS is a transaction. The actions of opening a file, writing to it, and closing the file encompass one transaction that is logged. In addition, other actions such as creating directories, and removing and renaming files are also transactions that are logged. In essence, every action that we want to appear atomic to the client and remain consistent is logged.

Notice that BSFS' transactions do guarantee *durability* for all completed transactions. This means that if data is written to a file asynchronously and then the system crashes after the file has been closed or synchronized, but before that data has been written to the database disk storage (via a checkpoint), that transaction can be recovered and replayed when the system is restarted. Completed transactions are not buffered in memory.

Also notice that this close-based transaction semantic allows BSFS to improve efficiency by buffering more of its data. This is similar to the open()/close() semantics of NFS, which state that file data might not have actually been committed to disk until a file is closed. BSFS uses this semantic to its advantage by: a) buffering contiguous writes so they are written in large chunks (NFS writes are no larger than 8k blocks); b) not committing a BerkeleyDB transaction to disk until a file is closed or synchronized (or a synchronous write occurs). Also, log space is saved by not logging individual writes.

Implementation

In this section, we will discuss some of the key implementation challenges that manifested from our design and the tools we chose to implement BSFS.

As stated in the design section, BSFS is implemented as a three-layer system: a network file system interface, a BSFS abstraction layer, and a persistent B-Tree implementation. All of these layers run together in one process as a single executable. BSFS utilizes *asynchronous programming techniques* to guarantee responsiveness of this single process while avoiding the complexity and deadlocking issues of threads. Each individual logical operation takes only a short amount of time; a lot of the framework for this is already provided by the file system interface we started with.

One of the three layers, the persistent B-Tree, complete with a transactional subsystem, is provided for us by Sleepycat's BerkeleyDB software. This section will discuss implementation of the other two layers in detail below.

SFS Network Interface - sfsusrv

The network file system interface is a modified *sfsusrv* (written by Frans Kaashoek). The sfsusrv, a user level Self-Certifying File System (SFS) server, provides an NFS3 interface to an underlying POSIX file system. SFS is "a secure, global file system with completely decentralized control", which, for the purposes of *sfsusrv*, provides an authenticated and decentralized package for writing user-level file systems in UNIX derivatives. For our purposes, sfsusrv is modified to call our B-tree based file system abstraction instead of the operating system's standard POSIX implementation.

The sfsusrv interface eases development, testing and demonstration of BSFS. In contrast, we could have implemented BSFS as a kernel module or as an application library, but with various disadvantages. Developing kernel modules is much more difficult than developing user space applications, because of greater chance of crashing the entire computer and weaker debugging facilities. Furthermore, developing in user space permits the use of other common user space libraries, such as Berkeley DB, which speeds development. Alternatively to using sfsusrv, BSFS could have been structured as an application library, but this would have greatly limited BSFS's generality as a file system and our options for testing it. The great advantage to having a file system as a data storage mechanism is its great ease of use and testing. For example, a file system can be accessed and tested using common shell commands, one of the advantages of using sfsusrv to implement BSFS as a standard file system.

There were not any real design challenges in modifying sfsusrv, although, some minor difficulties were encountered. The UID and GID handling has not been implemented in sfsusrv, and so BSFS is cannot check permissions completely, since it has no information on the user who is making the query. Still, UID and GID fields exist in the BSFS metadata, and they can be set and retrieved. When sfsusrv eventually implements UID and GID handling, it will be a simple addition to BSFS.

BSFS Abstraction Layer - B-Tree -> File system

BerkeleyDB directly provides an API to perform most of the basic functionality a filesystem needs. It can read/write key/data pairs of arbitrary length. It can cleanly remove records. It provides a mechanism to perform partial gets/puts on records, as well as extending and shrinking the data portion of a record. By wrapping this API with some semantic checking (i.e. duplicate file names are not allowed, users cannot manually create or remove special files like '.'), it was not difficult to implement basic file operations in BSFS. All of these file read/write/open/close commands occur in O(log n) time (where n is the total number of files in the system) and usually require only one physical lookup. Multiple lookups are needed only in a few cases: a) when pre-operation semantic checks occur (ex. making files or directories require validating the parent directory), b) when a logical operation operates on multiple records, which happens in making and removing directories (the special files "." and ".." must be added), and writing to a file (some information in the metadata, such as file size, must be updated, as well as the data itself). Even with these occasional extra lookups, BSFS performs most operations with a single O(log n) read or write from the Berkeley B-Tree, making BSFS much more elegant traditional UNIX filesystems without sacrificing efficiency. There is no need to manage lists of free disk blocks or any of the other complexities in a traditional file system; these are handled by BerkeleyDB's internal record-storage code.

Although we were able to implement BSFS' basic file I/O design very straightforwardly in BerkeleyDB, more advanced functionality posed some challenges. This section will discuss the following further implementation challenges: *inode numbering*, *metadata*, *directory listings, transactions,* and o*ptimizations.* Finally, this section will conclude with a discussion regarding where and why BSFS breaks traditional UNIX filesystem semantics.

Inode Numbering

One difficult issue in using BerkeleyDB to implicitly give files storage space is the assignment of *inode numbers*. NFS servers are required to identify each file by a unique, consistent number. This falls nicely out of traditional Unix file systems, where *inode numbers* are a basic part of the inode table. BSFS has no

inode table and no real concept of inodes, only metadata associated with keys. Therefore BSFS has to pick a unique 32-bit integer for each created file. To implement this, we decided on a simple incrementing number. This, unfortunately, means the file system is limited to the creation of 2^32 total files, and when a file is deleted, that unique number is never recovered. However, statistical analysis shows that this number will not be exhausted for many years of average use. If a user creates 100 files per second, then they can create files for about 42949673 seconds, or 497 days, or about 1.36 years before they reach the end of the inode number counter. However, this is still a significant limitation that prohibits the long-term use of BSFS. A repacking utility would eventually be required, and could be written as a simple utility that renumbers all of the files in BSFS sequentially. This utility would take some time to run, as it would touch the metadata of every file in the system, but it would only need to be run once every few years.

Metadata- storage and structure

As stated earlier, each file contains two pieces of information: data and metadata. We elected to store these two pieces in separate database records, so each filename key will have two physical records associated with it. This provides a performance advantage over storing metadata in the same record as the data. BerkeleyDB can read/write full records faster than partial records, and is more likely to keep data for small records physically near the key than for large records, reducing the number of disk seeks to read/write metadata. (See the discussion on overflow records in the optimizations section below for further details.) Therefore, metadata, which is read or written every time a file is opened, closed, read, written, or stat'd, is stored in a single record separate from file data at a significant performance increase. Additionally advantageous is that the metadata and the first block of data are still stored physically near each other, so there is no performance downside.

The actual downside is an increase in complexity. BSFS utilizes BerkeleyDB's duplicate record interface to access the two records separately, and it has to open a cursor into the database (an enumeration of sorts) to walk from the data record to the metadata. However, we managed to obscure this complexity using read/write macros that leave the resulting code quite readable and uncomplicated.

We considered splitting the actual file data into duplicate records of a fixed chunk size, causing variable numbers of variable-length data records per file. However, preliminary results showed that inserting a record into the database is the single most costly database operation, and it is actually quicker to use a single record for large files, even though that record might require an extra disk seek to retrieve its information due to overflow pages (seek the optimization section below for more details).

Metadata is represented as a single C-style struct that is written directly to disk whenever file writes are committed. This struct is updated whenever a file's attributes are changed via the NFS setattr call, and also whenever the file's size and modification time changes as a result of writing to it. As file data is changed, the metadata must be updated to reflect access times and file size, and file permissions can be changed manually. BSFS can cache the metadata on a per-filehandle basis and needs to write it to disk at most once per transaction. It would be optimal to not require metadata updates when writes occur, but BerkeleyDB doesn't provide any functionality to automatically retrieve record size or access times, so BSFS must store it manually. However, the performance cost is not too bad, thanks to per-transaction caching and the BerkeleyDB's high performance with full-record writes for constant-sized records.

Directory listings

As discussed in the design section, BSFS stores directory structure implicitly in the B-Tree using a modified lexographical sort. BerkelyDB actually lets us directly define the sort function used to store keys, so this is straightforward to implement.

Also, BerkeleyDB allows one to open *cursors* into the B-Tree, which begin at the record most closely matching a given starting point and allow the programmer to iterate along the B-Tree in the sorted order of the keys. BSFS uses this functionality to generate directory listings on the fly. It opens a cursor to the directory name plus a forward slash, which will set the cursor to begin returning elements starting from the first element contained in that directory. (This is a nice trick, because we store directory names in the B-Tree without the trailing slash, so they won't set the cursor but will properly show up as listed in the

parent directory.) A directory always contains some files, because the dummy files '.' and '..' are created along with the directory. Thus, the following algorithm is used to generate a directory listing:

1) Create a Berkeley DB cursor and initialize it at the first key that begins with the directory name (and including the trailing slash).
2) Iterate across the database until a key is found with fewer or greater slashes than the last key, or has a terminal path element that is different from the previous terminal path element. At this point, the iterator has reached the end of the directory, thanks to the sorting function described above.

BSFS has a pathname resolver which makes '.' and '..' usually transparent. These dummy directories are automatically resolved away before files are opened and deleted, but readdir() does not resolve pathnames, it prints them as is, so '..' and '.' are correctly reported as present directories.

Directory files (without the trailing slash) contain the directory metadata and ensure the directory is listed in its parent directory.

These directory reads are very efficient, because they read blocks of metadata linearly in the B-Tree. Since metadata records are small enough that they are stored alongside their associated keys, BerkeleyDB caches the metadata for files in a directory automatically when the cursor is opened (and, in large directories, when the cache needs another page from the B-Tree). No extra disk seeks are necessary to iterate through the B-Tree's metadata, allowing efficient and elegant retrieval of directory information.

Transactions

A lot of the work in implementing *fault-tolerance* is given to us by BerkeleyDB. What remains to decide is what to turn into a transaction to be logged, and when to checkpoint.

Transactions are logged for several different file system operations. For example, file data is logged by beginning a transaction when a file is opened. This creates a unique transaction id, which is used with all the database writes. This id marks these writes as belonging with a certain transaction. Once the file is closed, this transaction is noted as being committed in the log. Other file system operations are logged in the same way that file data changes are logged. These operations include creating and removing a directory, updating file metadata, and removing a file.

We checkpoint the log every 30 seconds. This strikes a balance between checkpointing after every write, which would hinder logging performance, and never checkpointing, which would hinder recovery performance. Once a checkpoint is entered into the log, the database is guaranteed to be consistent with all the log entries up to that point, so the prior log files can be discarded.

Optimizations

Speed turned out to be the biggest difficulty in implementing BSFS. Although reads and directory listings are very fast, it turns out that writing to the database can be costly, especially for large files. Berkeley DB stores its data in database *pages*, and when a data item gets too big (typically >1/2 page) to fit comfortably inside a page, Berkeley DB relegates it to an *overflow page*. Every access to an overflowed data item then costs an additional page hit. One solution is to increase the page size, but the database starts growing rapidly when page size is large. This is a similar problem to increasing allocation-unit size in a typical file system – increasing page size increases accessibility and performance, but decreases storage efficiency. Also, the maximum page size BerkeleyDB supports is 64 kilobytes, which is not large enough to support large files in-page anyway. Preliminary testing showed that any page size between 16 and 64 kilobytes results in similar performance when BerkeleyDB's cache is sufficiently large.

Although the extra seeks for large-file writes does decrease efficiency, a single 10mb write takes about one second on our systems, which is not unreasonable. The more serious write-performance issues are successive small-file writes and synchronous writes, where every small update to a file must be flushed to disk, that occur in programs like gcc. To achieve more throughput, BSFS breaks standard NFS semantics for synchronous writes; see below.

Implementation Summary and Specific Changes to NFS Semantics

Overall, the BSFS Abstraction layer implements all basic functionality necessary to integrate it with sfsusrv, including: file read/write/create/delete/rename, directory create/delete/listing, and metadata operations.

BSFS' structure called for a few changes to POSIX semantics in order to maintain efficiency and elegance. Specifically, there were changes in 2 areas: 1) access permissions, 2) synchronous writes. The access permission bits have slightly different semantics due to the way the B-tree works: the parent directories' permission will not be checked when a subdirectory is listed or a file is opened. This is more open than normal Unix semantics because it exposes files and directories that might normally be protected by a parent directory's more restrictive permissions. This is due to skipping the lookups of all the parent directories when looking up files in the B-tree. The permissions on the file itself and on a directory would still be effective, so some security still remains. Also, the synchronous write behavior of NFS was disabled, because it lead to many small, unbufferable updates to the file data, which caused excessive log growth on the order of gigabytes of log space for megabytes of file writes.

Speed Performance

BSFS' design goal was to hit speed performance adequate for normal use, and not too much worse than a traditional UNIX filesystem. We wrote our own benchmarking scripts and ran them on NFS, FFS, and BSFS (all running on the same computer servers in MIT's Laboratory For Computer Science). BSFS' B-Tree was stored on a local disk rather than an NFS server to provide more realistic performance benchmarks.

Benchmarks tested writing and reading performance with a variety of file sizes: 10 b, 50 b, 100 b, 500 b, 1 Kb, 5 Kb, 10 Kb, 50 Kb, 100 Kb, 500 Kb, 1 Mb.

Writing Test

Since write times are dependent on other system loads, multiple timing measurements were taken and averaged. For the smaller file sizes (between 10 b and 10 Kb) the time elapsed for a single write was difficult to measure in the millisecond range, so the time to write 100 files of a certain file size was measured instead. The average of ten such measurements were averaged. For the larger file sizes (between 50 Kb to 1 Mb) the time to write a single file was measured, and the averages of a hundred such measurements were taken for each file size.

Reading Test

Similarly in the read tests, for 10b to 10 Kb file sizes, the time to read 100 files was measured and ten measurements were averaged. Testing the larger file size range involved measuring the time of a single file read and taking the average over a 100 tries. The reading test actually illustrates the caching behavior of the file system more than physical-read performance. In the test scripts, the files are written immediately before the measurements are taken. FFS and NFS keeps the file data in cache so read times are fast. Unfortunately, BSFS does not keep the data in cache after a write. As a result, the first read in the set usually takes much longer of BSFS (on the order of a few seconds). Improved read/write caching would improve BSFS' reading speed in cases like these.

Lookup Test

For this test, a directory with a large number of files was constructed. The lookup performance was a measure of how long it took to do an 'ls –l'. Because of caching effects, only the time of the first "ls -l" is of interest. Lookup times in directories with 10,000 files and 100,000 files were measured. BSFS lookup times seem to be as good or better than the other file systems.

Filesystem	10,000 files	100,000 files
BSFS	0.007 s	0.012 s
FFS	0.007 s	0.027 s
NFS	0.005 s	0.015 s

Figure 3 – Lookup Test Results

However, these lookup time numbers are suspect for several reasons. First, the times are small enough that the load on the test system caused great variation in the results. Second, FreeBSD was caching

directory entries, and it was hard to avoid this effect. Third, as a practical matter, the NFS time should not be better than the FFS time, since it uses FFS as its underlying file storage. In actuality, BSFS should have much better lookup times because of its B-tree structure. However, limitations on the number of files we could create, and overhead in the NFS protocol prevented us from proving this definitively.

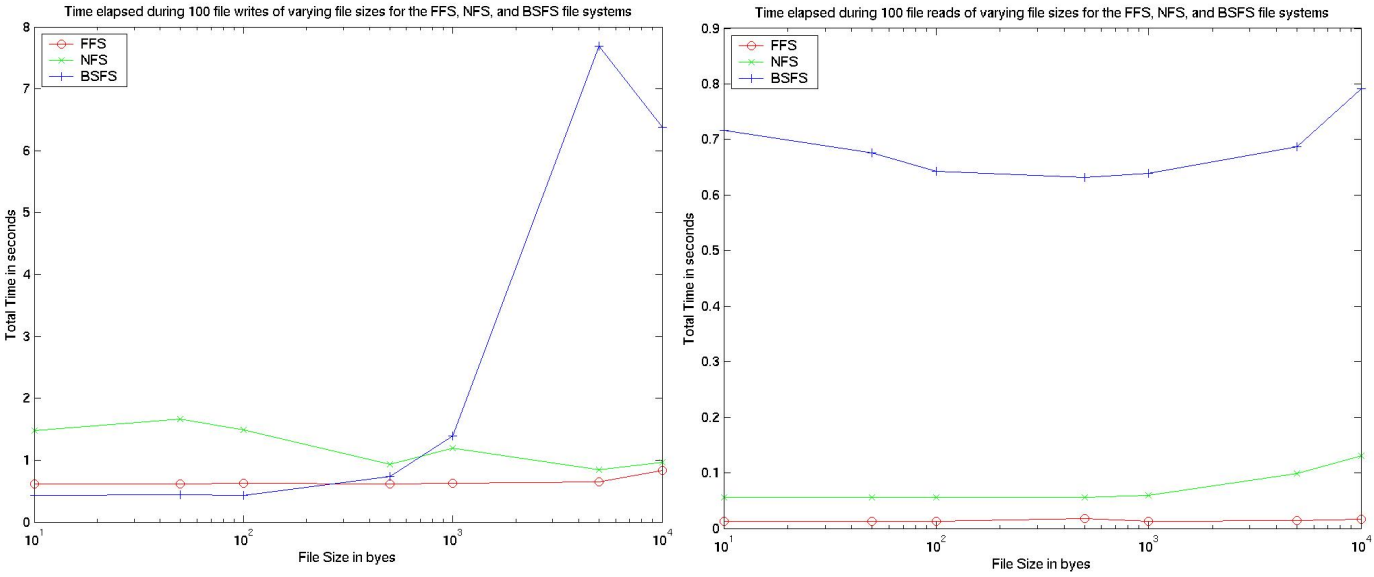

Figure 4 – Read/Write Test Results

Results

BSFS read time averages are much slower than the other file systems. On a closer look at the test outputs, the first measurement in the set of ten for small writes is on the order of a few seconds. This means that the 100 small files were not in cache after the file generation. However, the series of single large file read in BSFS does not exhibit the initial read cost, indicating that the issue is with BSFS's caching management of many files. The steady state read times of the following are still a few times longer than NFS and FFS, perhaps due to database overhead costs. BSFS write times grow exponentially with file sizes. This is due to the increased overhead of logging, which writes a copy of the entire record into the log both before and after each change that is made. Write times get exponentially worse because the amount of data being written to the log is growing exponentially as file size grows.

Generally, BSFS does not perform as well in writing and reading as the other file systems. Read performance suffered due to insufficiently clever caching, and write performance suffered both due to insufficiently clever caching and implicit performance costs in inserting and modifying keys in BerkeleyDB's B-Tree and log.

Future work

BSFS is still a work in progress. Possible future work includes fixing sfsusrv to pass along the UID and GID that is included with the NFS request, so that better permission checking can be implemented. In addition, BSFS could improve throughput either by improving caching or by being restructured to use the B-tree only for storing file inodes. This would require file data to be kept in some kind of block storage that is external to the database. This would greatly add to the complexity of the system, but would also provide more opportunities for optimizing its performance.

Summary

BSFS elegantly stores files directly in a B-Tree with low *complexity, fault-tolerance*, and *usable performance compared to existing file systems*. It elegantly represents files and metadata as records in the B-Tree, and improves efficiency by separating the storage of the metadata record from the data record; it efficiently generates directory listings on the fly from the implicit structure of the B-Tree; it achieves usable rename capability in a B-Tree that cannot rename keys; and it ensures consistency-level recoverability by using transactions. BSFS requires many fewer data structures than a traditional UNIX filesystem, and it hides its transactional functionality behind BerkeleyDB's API, obscuring its JFS-like complexity. It successfully recovered from a variety of crashes we invoked, while performing all kinds of database data and metadata operations. While meeting all these design goals, it still maintains usable performance, if markedly worse than FFS/NFS. Performance excels in file-lookup operations, and improved caching would almost certainly provide FFS-comparable read performance. Though throughput is not optimal, BSFS trades off performance for reliability and simplicity, striking a different performance balance than a traditional UNIX filesystem. Overall, BSFS provides a functional and elegantly implemented traditional file system augmented with effective fault-tolerance.

Related Work

BSFS is built on both the Self-Certifying File system (SFS), which provides authentication and network access to BSFS, and *sfsusrv,* a piece of Frans Kaashoek's code which provides a user-level SFS file system that implements the NFS3 interface.

B-Trees were previously used in the XFS file system, which replaces many of the bitmapped and table-based file structures in FFS with B+-Trees, due to their faster access time and greater scalability. However, XFS is still a traditional file-system in the sense that its B+-Trees mimic common data structures (inode table, directories, free lists, etc). BSFS has no need for these extra data structures, because storage of file records is managed under the hood in BerkeleyDB.

SFS RO implements a simple read-only file system using a hash-table that hashes each file's block. While somewhat similar in structure to the BSFS B-Tree, SFS RO's primary design concerns revolve around data integrity and security.

IBM's JFS is one of several journaling file systems which using logging and transactions to provide rock-solid crash recovery with fast startup time. BSFS improves on this concept by obscuring all the journaling/logging/transaction code underneath the BerkeleyDB package, so the filesystem implementer can implement correct transactional behavior more easily.

References

Berkeley DB programmer's guide. Sleepycat Software. 10 Oct. 2001 <http://www.sleepycat.com/docs/reftoc.html>.

Fu, Kevin, M. Frans Kaashoek, and David Mazieres. *Fast and secure distributed read-only filesystem*.

JFS Overview. Jan 2000. IBM. 27 Now 2001 <http://www-106.ibm.com/developerworks/library/jfs.html>.

SFS website. 20 Nov. 2001 <http://www.fs.net/sfs/new-york.lcs.mit.edu:85xq6pznt4mgfvj4mb23x6b8adak55ue/pub/sfswww/index.html>.

Sweeny, Adam. *Scalability in the XFS File System*. USENIX Winter 1996.