File System Extensibility and Reliability Using an in-Kernel Database

A Thesis Presented

by

Aditya Kashyap

to

The Graduate School

in partial fulfillment of the

Requirements

for the degree of

Master of Science

in

Computer Science

Stony Brook University

Technical Report FSL-04-06
December 2004

Abstract of the Thesis

File System Extensibility and Reliability Using an in-Kernel Database

by
Aditya Kashyap
Master of Science
in
Computer Science
Stony Brook University
2004

File systems are responsible for storing data efficiently and reliably. Databases also have similar tasks, but generally have more concern for reliability than file systems. We have developed a file system, KBDBFS, on top of an in-kernel database. KBDBFS supports standard file system operations, as well as other useful extensions such as extended attributes, ACLs, and a new operation to retrieve all the names of a file with hardlinks.

To implement KBDBFS, we have ported the Berkeley DB software into the Linux kernel. Berkeley DB provides efficient and persistent storage of key-value pairs using hash tables and B-trees with full ACID semantics. Berkeley DB is a stable, high-quality, fully-featured, and widely-used embedded database. KBDBFS's fundamental operations manipulate a database schema, rather than low-level objects like bitmaps and allocation tables, so it is easily extensible. KBDBFS uses BDB transactions around its file system operations; this ensures that KBDBFS operations are reliable. KBDBFS also exports the transaction API to user-level processes, allowing them to perform several file system operations atomically. Our performance evaluation shows that KBDBFS has acceptable overheads for user-like workloads.

To Mom, Dad, Abhi, Rajima, and Appa

Contents

List of Figures

List of Tables

Acknowledgments

I would first of all like to thank my advisor, Prof. Erez Zadok, for the constant source of help and encouragement he has been through my Master's program at Stony Brook. I would also like to thank Prof. Tzi-Cker Chiueh, and Prof. Alex Mohr for agreeing to be on my committee and for their valuable inputs.

This work would not have been possible had it not been for the many contributions from the crazy, but infinitely lovable band of people I've spent the past year and a half working with, more popularly known as the FSL. Many thanks to Chip, without whose coding skills, I would have given up kernel programming after my first *oops*. To D, for being a great friend when I needed it the most, and for letting me win at *Crack Attack*. To Mohammad, without whose inputs, this project wouldn't have been where it is now. To Sean, for all the geeky conversations, the endless rounds of laughter, and many late-night, unduly procrastinated, pre-project deadline, coding sessions. To Akshat, for all the *Pink Floyd* sing-alongs, and cribbing. To Avishay, with whom I could re-live my college-band days, for all the jamming sessions and performances. Finally, to everyone at the FSL, for making it the amazing place it is.

I would also like to thank the following people, who all contributed something to this project, from a simple bug fix to writing a component that used KBDB. Their input was valuable during the evolution of KBDBFS: Jay Pradip Dave, Nikolai Joukov, Harikesavan Pathangi Krishnan, Devaki Kulkarni, Abhishek Rai, Gopalan Sivathanu, Avishay Traeger, Charles P. Wright, and Mohammad Nayyer Zubair.

This work was partially made possible by an NSF CAREER award EIA-0133589, NSF award CCR-0310493, and HP/Intel gifts numbers 87128 and 88415.1.

Chapter 1

Introduction

Operating systems and databases run on the same machine, yet historically each usually ignored the other. The database system wants to access data directly, without the interference of an OS cache, so that it can provide good performance and strong reliability guarantees. Therefore, the database has its own replacement mechanisms, and usually allocates a large fixed block of memory. Unlike operating system caches, the database cache does not grow and shrink with available resources. Operating system developers also ignore databases: when they need to store data they roll their own solutions. This is less than ideal, because the state information operating systems store is often ancillary to their function. Rather than spending time developing OS functionality, developers spend time duplicating database functionality—often without the reliability and efficiency of the database.

Operating systems are complex pieces of software, which must manage large amounts of data. Yet traditionally, operating systems have used simple data structures like linked lists and hash tables. Only recently have more advanced structures, such as binary trees, become more common within commodity operating systems. It is even more difficult to develop persistent data structures within the kernel. Standard I/O mechanisms are not always available, and performing I/O on multiple regions of a file often requires strict ordering to avoid deadlocks. Worse yet, often kernel code simply ignores persistence, or depends on user-space to save and restore kernel parameters. Ideally, a kernel should have access to a suite of advanced data structures that provision for persistence, without changing the kernel itself.

On the other hand, database systems make use of advanced data structures designed to be fast, efficient, and persistent. Additionally, databases take great care to not only store data, but store it reliably. The most important primitive exposed by database systems to provide this reliability is a *transaction*. Transactions systems have four desirable properties: atomicity (a transaction is either applied or not), consistency (integrity constraints are not violated), isolation (issuing a series of concurrent transactions executes as if each transaction was executed in turn), and durability (once a transaction is completed any kind of system failure does not erase it). These properties are collectively referred to as the ACID properties.

Many database systems are large, requiring many resources, and consisting of both clients and servers. However, an *embedded* database runs inside the address space of a process. Instead of using expensive network operations to communicate with a database server, an embedded database simply makes function calls. Embedded databases do not need to parse SQL, formulate query plans, or manage stored procedures; this is delegated to the programmer. Because an embedded database is lightweight and designed to run in the address space of the client, it makes an ideal candidate for use in an OS kernel. One such database is the Berkeley Database (BDB). BDB is a well-known and trusted system used in many industrial-strength applications.

By embedding BDB inside of the Linux kernel, any kernel component can efficiently and reliably store data. We call our BDB port to the kernel *Kernel Berkeley DB(KBDB)*. One natural application of efficient and reliable data storage is a file system. We have built a file system, *Kernel Berkeley DB File System (KBDBFS)*, on top of our KBDB port. Building a file system on top of a database has three advantages. First, a file system built on a database is easily extensible. Second, databases have been designed for reliable storage, something that many file systems are not designed for. Third, file system management is made simple: normal user-level utilities can perform important management functions like backup. Next we discuss each of these features in turn.

A simple change such as adding an extra inode attribute for inode creation time would be difficult for most disk-based file systems. The kernel programmer would need to modify the on-disk inode structure, which is often of a fixed size. To add more flexible data, such as name-value pairs to a disk-based file system, programmers must deal with bitmaps, allocation tables, indirect blocks, and other complex structures. To make the same changes with KBDBFS, only the schema needs to be changed. For example, our file system supports extended attributes and ACLs. To add ACLs and extended attributes to Ext2 takes 2,564 lines of code, and the on-disk format is incompatible with a non-ACL enabled file system. To add extended attributes and ACLs to KBDBFS took 595 lines of code and one person-week (serialization of an ACL into an attribute, deserialization, and checking permissions based on an ACL structure were already handled by the kernel) . Additionally, extended attributes scale well because they use efficient B-trees to access data. KBDBFS imposes no artificial limits on the number of attributes or their size. Another simple extension to KBDBFS is that we keep track of all of a file's names, which allows easily mapping from an inode back to all of its hardlinks.

As the underlying database system improves, so will the file system—without major effort on the part of the file system developer. If KBDB algorithms improve in the future, then KBDBFS will become faster. One KBDB feature that we ported after writing KBDBFS was encryption. With only a single line of KBDB-related code, KBDBFS could add strong cryptography to the on-disk format. When dealing with code without system calls, porting code to the kernel is simple. For encryption, the only modification was to include it in the Makefile. KBDBFS was not designed with replication in mind, but BDB supports replication. When replication is ported from BDB to KBDB, then KBDBFS will support replication for free.

Most traditional file systems are simply best-effort storage. For performance reasons, file systems do not have the same ACID properties as a database: writes can be safely committed to disk, or may end up lost in memory (e.g., if the write call returns but a buffer has not yet been written, the buffer could be lost due to a machine crash). By building a file system on top of a database, we can come closer to achieving these ACID properties. At the same time, the file system code becomes simpler, because the database handles the work of underlying data storage. Because the file system code is smaller, it can be developed more rapidly and will have fewer bugs.

Database transactions are a useful tool for the file system. File system reliability and error handling is increased by atomicity. For example, the rename operation fundamentally requires three steps, any one of which could go wrong: (1) unlinking the new name if it exists, (2) linking the old name to the new name, (3) and removing the old name. By wrapping these steps inside of a transaction, the file system does not need to worry about handling these individual errors.

Transactions also provide durability: after performing an operation that is wrapped in a transaction there is no need to worry about the results of the operation being lost.

Some benefits of transactions can also be exported to user-space. For example, a sequence of `readdir` system calls can be wrapped in a single transaction. This would prevent create, rename, or delete operations from interfering with the listing of the directory. There are many applications that would benefit from ACID properties (e.g., electronic mail transfer and delivery), yet the OS can not provide them. KBDBFS comes a step closer to realizing this goal, but unfortunately OS caches interfere with BDB's management of the data. If an operation uses the database as an authoritative source, then it meets the ACID properties. If an operation uses OS caches as an authoritative source or modifies the file system, then only the durability and consistency properties hold. Isolation does not hold when using caches, because the caches can be accessed without the database being consulted. The database preserves atomicity for transactions, but if the transaction is aborted, OS caches may be in an inconsistent state. In this case we can flush as much file system cache state as possible (actively used objects can not be removed, but we ensure that actively used objects are no longer used). Supporting the full ACID properties requires modifying much of the VFS and OS caching infrastructure, and is beyond the scope of this paper.

The transactions that KBDBFS provides are useful. For example, Sendmail's `mail.local` delivery agent appends messages to a mailbox. If the appends fail, then `mail.local` truncates the mailbox to its original size. With KBDBFS, all of the appends can be wrapped in a transaction. If one fails, then the ones that have been already applied have no effect.

For a file system to be truly useful, it must have both the in-kernel component and also user-space management utilities for initializing, recovering, and backing up file systems. By selecting BDB as a database storage engine in the kernel, we automatically have a user-space storage engine that is compatible. Instead of duplicating code to read and maintain file-system state within utilities like `fsck` and file-system–specific `dump` utilities, we can simply read the databases from user-level. BDB stores its databases as plain files within another file system. There is a small trade-off between performance and compatibility. It is slightly faster to access a raw device than an on-disk file, but in practice for data operations (i.e., read and write) disk I/O dominates regardless of whether a file system is used.

We believe that file systems built on top of in-kernel databases hold significant promise. New and innovative features can be rapidly prototyped, and stronger file system semantics can be explored. KBDBFS consists of 8,464 lines of code, whereas Ext3 consists of 14,814 lines of code (including the journalizing block device support, which is only used by Ext3). KBDBFS was also rapidly developed: it took only 7.5 person-months to develop the file system with all of its features.

The rest of the paper is organized as follows. Section 2 presents a high-level view of the Berkeley Database and its components, and details changes we made to run it within the Linux kernel. Section 3 describes the design of our file system based on KBDB. Section 4 presents a performance evaluation of our file system. Section 5 describes the applications developed using KBDB as a reusable operating system component. Section 6 discusses related work. We conclude in Section 7 and discuss future directions.

Chapter 2

The Berkeley Database

The Berkeley DB (BDB) is an efficient embedded database library that is portable across several operating systems. As a library, BDB runs in the address space of the application: no IPC or networking is required to use the database services. For these reasons, BDB is an ideal candidate to port to the kernel.

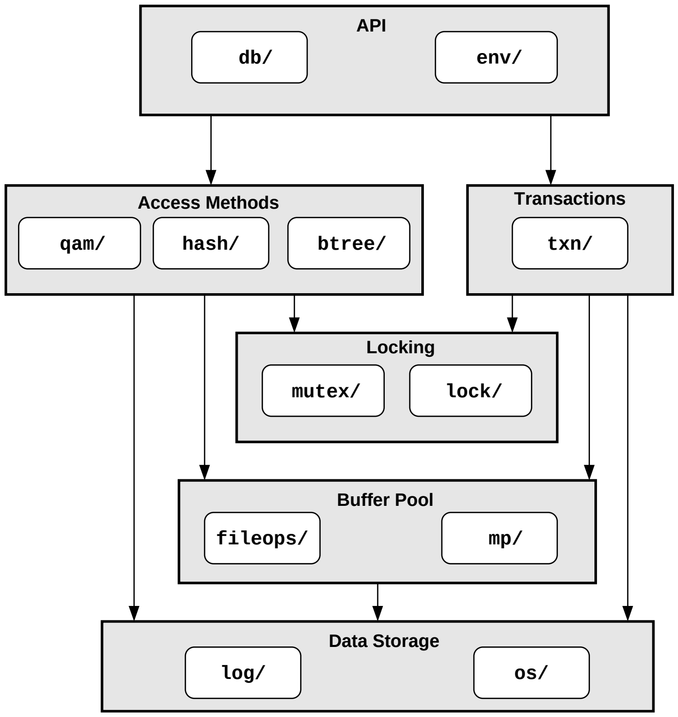

Figure 2.1: In-Kernel Berkeley DB Architecture

In-Kernel Berkeley DB Architecture

Figure 2.1 shows the components that we identified and ported to the Linux kernel to build a fully-functional in-kernel database that has support for BDB's core functionality. In our first prototype, we did not include replication, RPC, cryptography, and other useful, but non-essential features. The components have been grouped by their functionality. The arrows indicate the components' calling convention.

An application accesses the database using the APIs provided in the `db` and `env` directories.

The `db` directory defines the database API, which is used to configure, create, and open databases; insert, remove, and retrieve records; and iterate over database records using *cursors*. With cursors, applications can iterate over all records in a database starting from a specified key. This is particularly useful when databases have multiple data items per key (i.e., duplicate keys), or related keys that are adjacent in the database schema. The `env` directory defines the *environment* API, which is used to perform cross-database configuration and communication. Environments are required for locking and transactions.

We have ported three database access methods to KBDB, which are in the `btree`, `hash`, and `qam` directories. The `btree` and `hash` methods implement Btrees and hash tables, respectively. These data structures provide efficient key-value pair access. The `qam` method provides a queue access method that can be used by producer-consumer queues.

The transaction component provided in the `txn` directory allows a set of database operations to be performed atomically (even across multiple databases). Critical sections that require atomic operations can use the locking facilities in the `lock` directory.

Berkeley DB allocates a large buffer pool of memory during initialization, for storing in-kernel memory images of the database. All transactions and queries access the memory provided in the `fileops` and `mp` directories of the buffer pool component. The in-kernel memory image is copied onto a physical storage device periodically using the data storage methods in the `os` directory. All the components record the database operations in log files using the methods in the `log` directory. The data storage component is responsible for translating database operations like `put` into file system operations like `write`. The database `sync` method commits the data to the backing store.

Table 2.1 shows the effort involved in porting BDB into the Linux kernel version 2.4.24. We ported twelve essential components of the user-level Berkeley DB package. The complexity of the porting ranged from trivial to complex code changes. Most of our changes were isolated to header files and the `os` directory, which contains operating-system–specific methods (e.g., `open`, `readdir`, and `malloc`). Trivial changes included wrapper functions for routines like `stat`. Complex code changes included ensuring integrity with concurrent accesses to the database by more than one process or kernel thread.

The `os` directory involved a significant porting effort. There were two major changes: (1) we used kernel functions instead of system calls, and (2) we had to support memory allocation. We used the Linux kernel equivalents for system calls, changing the calling conventions as needed (e.g., `read` becomes a `filep->f_op->read` method, `rename` becomes a `vfs_rename`, etc.). BDB allows multiple processes to access the same database in memory. In user space this is done with shared memory. Processes, however, operate differently in the kernel and there is only one address space. Shared memory concepts in user-space had to be carefully examined in the context of the kernel address space. We ported memory allocation to the kernel level, using in-kernel shared memory, by creating an emulation layer that returns memory backed with `kmalloc` or `vmalloc`. `kmalloc` uses the slab allocator to allocate memory, whereas `vmalloc` uses the page allocator to allocate memory in pages that can be swapped out.

In addition to porting the native Berkeley DB code, we used certain utility functions that are available only in user-level libraries or the core kernel. We used long division from `libgcc`; `qsort` and the random number generator from `glibc`; and `ctime` from `dietlibc` [22]. Finally, we copied a few pathname-resolution functions from the kernel's `fs/namei.c`, since they were not exported to modules; this way our KBDB module could work with any unmodified Linux 2.4 kernel.

File or Directory	**Add**	**Del**	**Chg**
`db/`	4	-	2
`env/`	3	-	8
`qam/`	-	-	-
`hash/`	-	-	-
`btree/`	-	-	-
`txn/`	-	-	-
`mutex/`	34	1	6
`lock/`	42	10	92
`fileops/`	-	-	-
`mp/`	-	-	-
`log/`	-	-	-
`os/`	528	10	370
`dbinc/`	71	-	55
`db.h`	208	7	44
Total = 1,495	890	28	577
Inode namespace functions	2,131	-	-
Shared memory operations	640	-	-
Assorted wrapper functions	449	-	-
Quicksort	405	-	-
Long division	419	-	-
Random number generator	155	-	-
Grand Total = 5,694	-	-	-
Final Code Size = 155,789	-	-	-

Table 2.1: Porting effort of BDB to the Linux Kernel

Number of lines of code added, removed, or changed while porting BDB to the Linux kernel. A dash indicates no changes.

The total number of lines of code in our port of BDB in the Linux kernel is 1,495, which is less than 1% of the total lines of code in Berkeley Database, which is 150,095. Hence, by modifying only a few lines of code, we have a highly efficient, embedded database in the Linux kernel.

Chapter 3

Design

We describe the design of the Kernel Berkeley DB File System (KBDBFS) in this section. KBDBFS uses KBDB as a backing store for efficient and reliable data access. It makes use of the transaction subsystem in KBDB so that if file system operations wrapped within transactions fail midway, error recovery becomes easier. This way, when updating several databases, one failure does not result in a corrupted file system.

Figure 3.1 shows the architecture of KBDBFS. The file system contains two primary components: the file system module KBDBFS and a pseudo-device driver that interfaces with the buffer cache. Both components use the KBDB module to access their databases.

KBDBFS registers itself as a file system with the VFS and uses the KBDB interface to access the on-disk databases as required. The databases are essentially a backing store that record information as key-value pairs. The database files can exist on any file system that provides standard read-write capabilities.

The KBDBFS design is sub-divided into three parts. We first discuss the database schema in Section 3.1. Section 3.2 describes how we use transactions to enable ACID properties in the file system. User-level tools to manage the file system are described in Section 3.3.

3.1 KBDBFS Database Schema

KBDBFS is structured like a standard UNIX file system with directories, inodes, and data blocks associated with each inode. KBDBFS also supports hard links, symbolic links, extended attributes, and access control lists. As shown in Table 3.1, six databases are defined to store the file system schema.

- `lookup.db` stores directory entries,
- `meta.db` stores inode meta data,
- `block.db` stores data blocks,
- `symlink.db` stores symlinks,
- `hardlink.db` stores a mapping of child inode number to parent inode number and file name, and

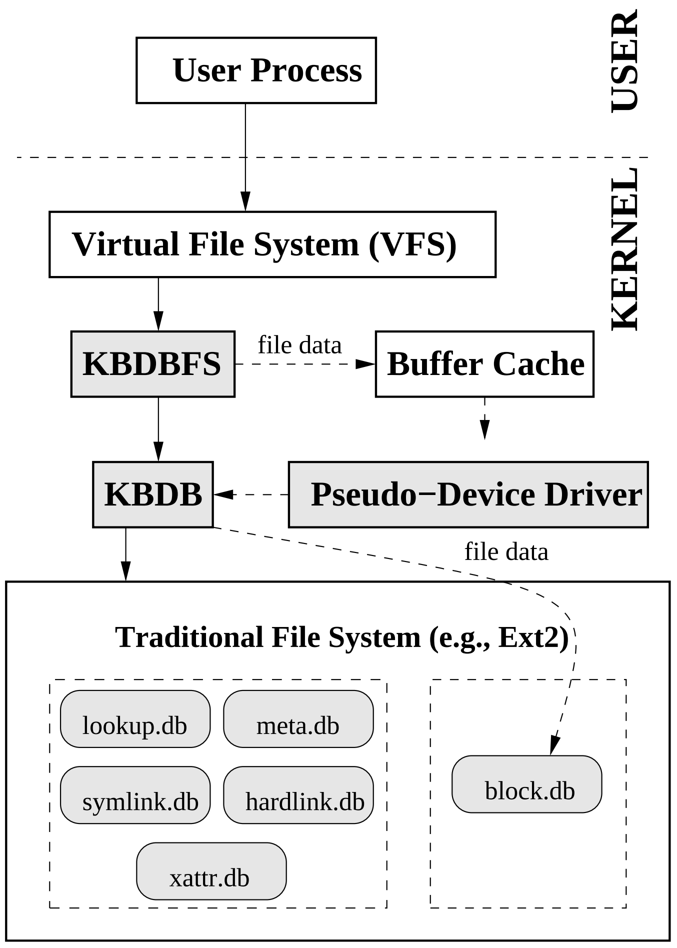

Figure 3.1: KBDBFS Architecture

KBDBFS Architecture

Database	**Key**	**Value**
`lookup.db`	Parent inode number || Child's name	Child inode number
`meta.db`	Inode number	Inode meta-data (e.g., size, atime, owner)
`block.db`	Inode number || Block index	Block's data
`hardlink.db`	Inode number	Parent inode number || Name
`symlink.db`	Inode number	Symlink value
`xattr.db`	Inode number || EA name	EA value

Table 3.1: KBDBFS Database schema.

KBDBFS Database schema.

- `xattr.db` stores ACLs and extended attributes.

All the databases in KBDBFS use the B-tree access method to store data, which orders stored keys. Using an appropriate comparison function improves data locality, and hence performance. Each database has a specific key-comparison function that determines the order in which keys are stored and retrieved.

Next, we describe each database. In Section 3.1.1 we describe the lookup database, followed by the meta and block databases in Sections 3.1.2 and 3.1.3, respectively. We describe the symlink database in Section 3.1.4, the hardlink database in Section 3.1.5, and the extended attribute and ACL database in Section 3.1.6.

3.1.1 Lookup Database

The lookup database, `lookup.db`, maps directory name information to inode numbers. The key used is a structure with two values: the parent inode number and the child file name. This is similar to a standard UNIX directory that maps names to inodes. We differ from UNIX directories in that UNIX directories are simply files with a special mode bit, but KBDBFS directories use an entirely different structure. Therefore, our directory locality does not rely upon the usual file system block allocator, but rather is maintained by the B-tree splitting algorithm. Interestingly, this structure more closely mirrors OS caches, which separate the directory-name-lookup cache from the page and block caches.

The compare function for the lookup database sorts keys first by the parent inode number and then by the child file name. On LOOKUP, KBDBFS uses the full key structure to uniquely access the entry in the lookup database. Because the database uses a B-tree, lookups are a sub-linear operation. One common task is to lookup each file in a directory; because files in the same directory are sorted together, locality is improved. Similarly, a single READDIR operation is also efficient as all entries in a directory are stored close together. The READDIR requires a cursor to be set on the directory inode number whose files need to be listed. The listing ends as soon as an entry with a different directory inode number is returned. Object creation operations, which include `create`, `mkdir`, `symlink`, and `link`, cause a name-inode pair to be inserted into the lookup database. This entry is removed when the corresponding removal operation is called for the name (`unlink` or `rmdir`).

3.1.2 Meta-Data Database

The meta-data database, `meta.db`, maps inode numbers to `stat(2)` information (e.g., atime, owner, and size). This database is queried on lookup to populate the in-memory inode fields. When an object is created, we insert an entry into this database. When an object is deleted, we remove the entry from this database. Directories are deleted immediately upon the `rmdir` operation, but open files are lazily deleted. After the file's name is removed, the inode must still exist until all open instances are closed, at which point it should be deleted. The VFS calls the `delete_inode` method when a file with no links is closed for the last time. If the machine crashes after the name is removed, but before the inode is deleted, then our `fsck` removes the inode from the `meta.db` database. KBDBFS updates its on-disk inodes asynchronously. When an inode field is updated, the inode is marked dirty. Periodically, the kernel calls the write method on these dirty inodes. At this point, KBDBFS replaces the old inode metadata in the database with the inode's new metadata.

We assign inode numbers in a monotonically-increasing order and there is no upper limit on the number of inodes we can support. To determine an inode number for a newly created file, we query the meta-data database once to fetch the highest used inode number. Once we fetch this value, each subsequent call to this method causes the new inode number to be atomically incremented by one. To get better data locality, and as a result, make finding the highest used inode number an efficient operation, the compare function for the meta-data database orders keys by the inode number of the file. The usage of B-trees in the meta-data database makes finding the highest used inode number efficient.

3.1.3 Block Database

The block database, `block.db`, maps unique block identifiers to actual blocks of data. The block identifier consists of the file's inode number and the block index. The block size for KBDBFS defaults to 4KB, the page size on an i386 platform. KBDBFS is unique in that it is built on a system that maps arbitrary keys to arbitrary data rather than a device which maps logical block numbers to fixed size data blocks. Because KBDBFS uses unique block identifiers, the design can be extended to use a compare-by-hash technique instead of a block identifier based on the inode number and block index [18]. KBDBFS handles sparse files too, because if an object is not present in the database, it is assumed to be zero-filled.

The compare function for the block database orders the keys first by inode number and then by the block index. This improves the common case of sequential file access. To achieve better interleaving of CPU and I/O operations, KBDBFS uses the buffer cache to store data before writing it to the block database. KBDBFS uses a pseudo-device driver built into the file system to achieve this. The driver registers itself when KBDBFS is mounted with a dynamically-assigned device number that is active as long as the file system is mounted. The device driver is called asynchronously by the buffer cache mechanism under two circumstances: the first being when the kernel flushes dirty buffers to disk, and the second when the file system requires a buffer that is not present in the buffer cache. The driver either writes data to or fetches data from the block database using the KBDB module. Using the O_SYNC option with `open(2)` provides durable writes, while also having the advantage of writing to a database log rather than perform potentially random synchronous operations. Entries from this database are removed when an inode is deleted. BDB also supports group commit, which can transparently combine multiple writes to the log file. This improves performance, and the application does not need to do anything special to allow for it.

3.1.4 Symlink Database

All symbolic-link related information is stored in a separate database called `symlink.db` so that the block database does not need to be queried to resolve symbolic links. This improves database cache locality and performance, especially for systems that use symlinks heavily in automounted pathnames [16, 24]. The symlink database maps the inode number of the link to its text. The symlink compare function orders keys by the inode number of the symlink.

3.1.5 Hardlink Database

Conventional Unix file systems have failed to efficiently solve the problem of mapping inode numbers back to full file names, especially for hard-linked files that have multiple names. The only way to solve this in existing FFS-like file systems requires a full traversal of the file system. HURD file systems provide a native operation that can retrieve all the names of a hard-linked file [3], but do not provide a standard Unix vnode interface and were implemented at the user level and hence suffered from poor performance.

The hardlink database is a unique addition to KBDBFS that aims to solve this problem. This database stores a mapping from the inode number to the directory information of the file: the parent inode number and the file name. Thus, in KBDBFS, enumerating all instances of a hard-linked file in the file system is just a matter of querying the hardlink database through the KBDB module (we provide an `ioctl` for this). Text editors and many other applications use `rename` to atomically replace a file. But this procedure tends to break hard links. The hardlink database can be used to easily restore these hard links.

The hardlink database maps an inode number to the parent directory and the name within that directory. To reconstruct the names of an inode, first the inode number is fetched from the hardlink database. This provides a single component of the pathname. Next, the parent inode is fetched from the database, until we reach the root of KBDBFS. KBDBFS has a mount time option to select one of the two modes of operation for the hardlink database.

The default mode (KBDBFS-HL) stores an entry in the hardlink database only when a file has multiple links associated with it. This mode can be used to keep track of files with multiple links, and therefore reduce the impact of this database on performance, because most files have a single name. Directories are always stored, because to reconstruct the names of an inode you must walk upward through the tree. Directories always have multiple links ("." and the link from their parent), so KBDBFS-HL is consistent in that any inode with multiple links exists in the hardlink database.

The second mode (KBDBFS-ALL) adds an entry to the hardlink database every time a file is created. This option can be used if the user wishes to find a file name given an inode number for any file in the file system.

The compare function for the hardlink database orders the keys according to the inode number of the file. Entries in this database are created on `create`, `mkdir`, and `link` operations for the KBDBFS-ALL mode and only on `link` and `mkdir` for the KBDBFS-HL mode; entries are removed on `unlink` and `rmdir` operations.

3.1.6 ACLs and Extended Attributes Database

Extended Attributes (EAs) are arbitrary name-value pairs associated with files and directories. They can be used to store system objects like capabilities of executables, a file's Access Control List (ACL), MIME type, etc.

The standard Unix permission model has three sets of permissions associated with a file: the owner, group, and all other users. This simple model is implemented using nine bits for each object. This permission model is adequate for many scenarios. However, for more complex scenarios that require access control with finer granularity, a number of workarounds are required to circumvent the limitations of this permission model. ACLs have been designed to overcome these limitations [6]. They allow assignment of permissions to individual users or groups, even if they do not correspond to the owner or the owning group.

EA and ACL support is now offered on most Unix-like systems. Solaris, FreeBSD, Irix, and HP-UX have separate ACL system calls. Linux uses the EA system calls to pass ACLs between the kernel and user level. However, each file system that supports EAs and ACLs has its own implementation with specific data structures that makes it incompatible with other implementations. To store EAs efficiently and save space, most file systems use different storage mechanisms.

Currently some file systems impose restrictions on the number of EAs or the size of each individual attribute that can be associated with a file. Ext2 and Ext3 have a limitation of one disk block size to store all EAs or ACLs for a single file. JFS, XFS, and ReiserFS, on the other hand, do not have the limitation of one disk block to store all EAs associated with a file, but use different mechanisms to store EAs efficiently. For example, XFS uses an elaborate mechanism to store EAs. EAs are stored directly in inodes if they are small. Medium sized EAs are stored as leaf-blocks of

B+ trees and large pairs are stored in full B+ trees [1]. JFS stores all EAs of an inode in a consecutive range of blocks on the file system [10]. ReiserFS stores EAs in a directory with a name that is created from the unique inode number of the file. These directories are stored in another directory that is hidden from the file system namespace. Inside the inode-specific directories, each EA is stored as a file whose name is the name of the attribute and the content of the file is the value of the attribute [11]. XFS, JFS, and ReiserFS limit the size of each individual attribute to 64KB. Although these mechanisms allow efficient storage of EAs, they make the implementation complex and file-system specific. This makes porting the code to other file systems difficult. Apart from this, fixing bugs or adding additional features is tedious and has to be done for every file system that supports EAs and ACLs.

KBDBFS uses a separate database as a backing store for EAs and ACLs. The key-value pair schema of the database made it an intuitive design to store EAs and extend EA and ACL functionality for KBDBFS. Our schema consists of a single database called `xattr.db` to store the EAs. This database maps an inode number and a name to the value of the EA. We chose to use B-trees as the data structure to organize the EA data. The B-tree structure allows us to efficiently retrieve a given extended attribute, and also list all attributes for an inode using a cursor.

Extending EA and ACL support was easy to add and debug for KBDBFS. Efficiency was handled by the already well-designed and scalable KBDB data structures. The added advantage of using the database is that a large number of EAs can be supported. There are no arbitrary limitations on the number or size of EAs or ACLs.

In conclusion, KBDBFS provides EA and ACL support with a simple design while offering a powerful feature.

3.2 Transactions

Each file system operation in KBDBFS is protected within a transaction. Transactions in KBDB are ACID, and thus enable ACID properties for a single file system operation in KBDBFS:

1. File system operations are **atomic**, and hence updates to multiple databases do not result in a corrupted file system if the operation fails midway. An error in updating any of the databases will cause the entire transaction to be aborted, thus ensuring a safe file system state.

2. File system operations are **consistent**, thus after a transaction is either committed or aborted, all the file system databases are in a consistent, non-corrupted state.

3. File system operations are **isolated**, so one operation does not interfere with another concurrent operation. Presently, the VFS already provides isolation for a single file system operation through directory and inode locking. Unfortunately, there is no way to disable the VFS locking, so for a operation KBDBFS pays twice: once for VFS locking and then again for database locking.

4. File system operations are **durable**, hence once the transaction is committed, the data is never lost. If the system crashes after the transaction is committed, then the operation is either safely in the transaction log, or will have already been committed to the database itself.

Transactions in KBDBFS can also be extended to applications that run on it. The file system exports three transaction ioctls:

- TXN_BEGIN associates a transaction with the process. All subsequent file system operations performed by the process are included in this transaction.
- TXN_COMMIT applies the transaction associated with the process to the database. The resources associated with the transaction are freed, and the process can begin a new transaction.
- TXN_ABORT destroys the transaction associated with the process, and nothing is committed to the database.

Applications can use this API to start a transaction before a certain set of operations, and depending upon their return values, choose to either commit or abort the transaction. For example, if the application is updating the `/etc/passwd` file and fails midway, without transactions, there would be a partially-written password file. If the above operations were wrapped within a transaction, the transaction would be aborted, and the file would not be partially modified.

Unix programs inform the system of the success or failure of their execution through their exit status. An exit status of zero means that the program succeeded, and a non-zero exit value means it failed. If the transaction is not committed or aborted through an `ioctl`, then it must be cleaned up when the process exists. When a transaction is begun, KBDBFS registers an on-exit callback with the transaction ID [23]. If the process has an exit status of zero, the transaction is committed. Otherwise the transaction is aborted.

When a transaction is aborted, the databases involved in the transaction are rolled back to their previous safe states, but the various system caches could still contain the incorrect, stale data. Under those circumstances, we flush the file system caches to remove all the stale data. We can not remove actively used objects, so we disconnect them from the file system tree; this prevents new users from being spawned. Once the objects are no longer used, they are removed from the cache.

If a user-level application cannot be modified to make use of the transaction API provided by KBDBFS, an LD_PRELOADed library can be used. Wrapping existing applications in the LD_PRELOADed library provides limited transaction support to legacy applications running on KBDBFS. We override the `open(3)`, `close(3)`, `read(3)`, and `write(3)` library calls. The first `open(3)` of a file on KBDBFS starts a transaction, and the last `close(3)` of a KBDBFS file commits the transaction. This is similar to the model of a session used in the Elephant File System [19]. If a `read(3)` or `write(3)` operation fails, then the transaction is aborted. This idea can be easily extended to other file system operations like `rename` and `unlink`.

3.3 User-Level File System Management

The database files created by the KBDB module use the same format as the user-level BDB library. Thus, the file system utilities that operate on the six KBDBFS databases can be simple user-level programs that make use of the BDB library. KBDBFS has an `mkfs` program that initializes the six databases to a consistent, empty initial state.

A `populate` program reads a pre-existing directory structure and creates the corresponding KBDBFS databases. The `populate` tool provides an easy method of migrating existing file systems to KBDBFS, or can be used for restoring a traditional backup. Again, just like in the kernel, it is easier for programmers to deal with a database schema rather than the low-level data structures traditionally found inside of a file system. A file system backup operation for KBDBFS now simply reduces to *copying* the six KBDBFS databases to a different location.

We also provide a `fsck` tool that verifies the integrity of the databases. It first opens the databases in recovery mode, to let BDB perform any necessary repairs. Next, it searches for orphans and removes them. Because BDB supports concurrent access from multiple processes, `fsck` or other file system tools can actually run *while* the file system is mounted.

Chapter 4

Evaluation

We conducted all tests on a 1.7Ghz Pentium 4 with 1GB of RAM, a 20GB Western Digital Caviar IDE disk, and a 200GB Maxtor IDE disk. The operating system was Red Hat Linux 9 running a 2.4.24 kernel with a small kernel patch that associated private data with each task structure [23]. To ensure a cold cache, we unmounted all involved file systems between each test. We ran all tests at least five times and computed 95% confidence intervals for the mean elapsed, system, user, and wait time using the Student-t distribution. In each case, the half-widths of the intervals were less than 5% of the mean. Wait time is the elapsed time less CPU time used and consists mostly of I/O, but process scheduling can also affect it.

KBDBFS with transaction support is semantically closer to Ext3 as a file system, with meta-data journaling enabled. Hence, we compare our results against Ext3 with meta-data journaling. It is slightly faster to access a raw device than an on-disk file, but in practice for data operations (i.e., read and write) disk I/O dominates regardless of whether a file system is used. BDB (and hence KDBD) stores its databases as plain files within another file system. It is slightly faster to access a raw device than an on-disk file, but in practice for data operations (i.e., read and write) disk I/O dominates regardless of whether a file system is used. For random reads, Ext2 is within 1% of a block device. Writes are asyncrhonous for both block device and Ext2, and thus return almost immediately. We use Ext2 as the underlying file system to store the six KBDBFS databases because the reliability that Ext3's journaling would extend is provided by the KBDB databases. We use a separate device, the Maxtor IDE disk to store the journal on Ext3, and as the log device for KBDBFS with transactions enabled. In order to overcome the ZCAV effect, the test partitions were located on the outer cylinders of the disk and were just large enough to accommodate the test data [4]. We ran KBDBFS in two test modes:

- KBDBFS-NO-TXN (KBDBFS with transactions disabled) and
- KBDBFS-TXN (KBDBFS with transactions enabled).

The KBDBFS-NO-TXN mode helps us identify where the overheads in the system come from. We tested KBDBFS with three different types of benchmarks:

- Am-utils: A compile benchmark that tests the system on a CPU-intensive, user-like load.
- Postmark: An I/O intensive benchmark that tests the worst-case I/O performance of the file system.

- To evaluate extended attributes and ACLs on KBDBFS, we used a suite of microbenchmarks based on similar tests made by Grünbacher [6].

Am-utils As a CPU-intensive benchmark, we unpacked, configured, and built Am-utils 6.1b3, which contains 430 files, 26 directories, and over 60,000 lines of C code. The build process begins by running several hundred small configuration tests to detect system features. It then builds a shared library, ten binaries, four scripts, and documentation. The Am-utils build process is CPU intensive, but it also exercises the file system because it creates a large number of temporary files and object files. We used Tracefs [2] to measure the exact distribution of operations. For this benchmark, 25% of the operations are writes, 22% are lseek operations, 20.5% are reads, 10% are open operations, 10% are close operations, and the remaining operations are a mix of readdir, lookup, etc.

Figure 4.1 shows the comparison between results of the Am-utils benchmark on Ext3 and KBDBFS in two modes: KBDBFS-NO-TXN and KBDBFS-TXN.

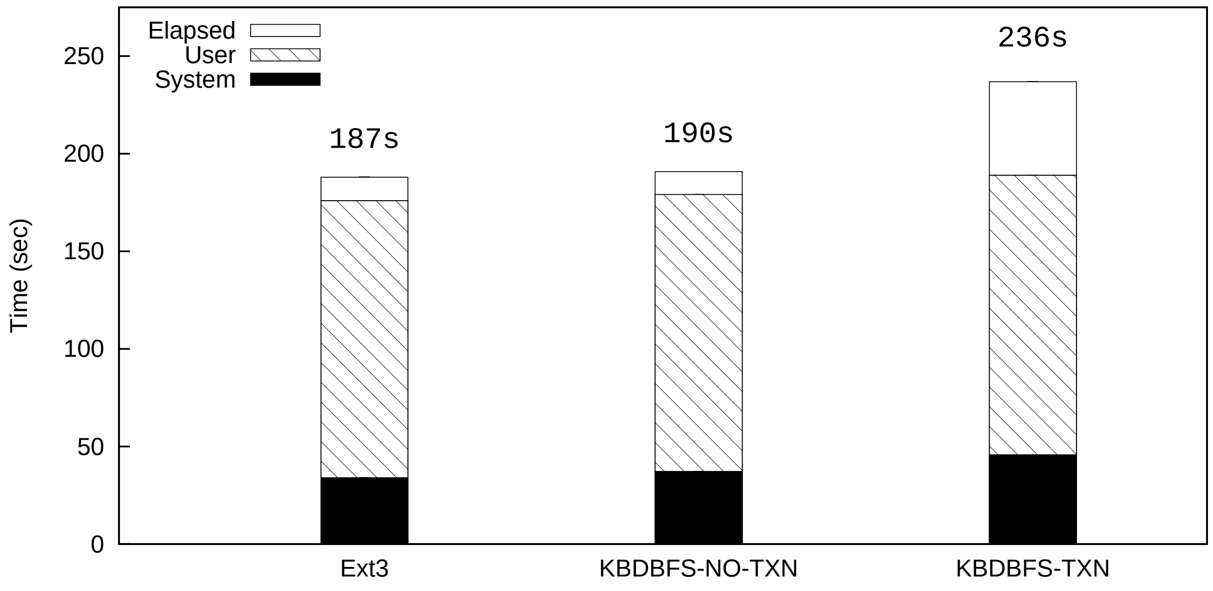

Figure 4.1: Am-utils benchmark results.

Am-utils benchmark results.

The Am-utils compile benchmark shows that KBDBFS with transactions disabled is almost equivalent to Ext3, with a 1.6% overhead in Elapsed time and a 9.5% overhead in system time over Ext3. KBDBFS with transactions enabled has a 26.2% overhead in Elapsed time and a 34% overhead in system time over Ext3. As expected, most of the overhead is in I/O time. The increased I/O is due to the transaction subsystem writing first to a log file, and then to the database when transactions are checkpointed. Although Ext3 writes meta-data to the journal, its data is written directly to the disk.

Postmark As an I/O-intensive benchmark we ran Postmark [9]. Postmark stresses the file system by performing a series of operations such as directory lookups, creations, and deletions on small files. Postmark has three phases:

- The file creation phase which creates a working set of files,
- The transactions (note that transactions is a term used by Postmark, and is distinct from KBDBFS transactions) phase, which involves creations, deletions, appends, and reads, and
- The file deletion phase removes all files in the working set.

We configured Postmark to create 5000 files (between 512 bytes and 10KB) and perform 20,000 transactions. A large number of small files is common in electronic mail and news servers where multiple users are randomly modifying small files. Figure 4.2 shows the results of Postmark on Ext3 and KBDBFS in two modes: KBDBFS-NO-TXN and KBDBFS-TXN.

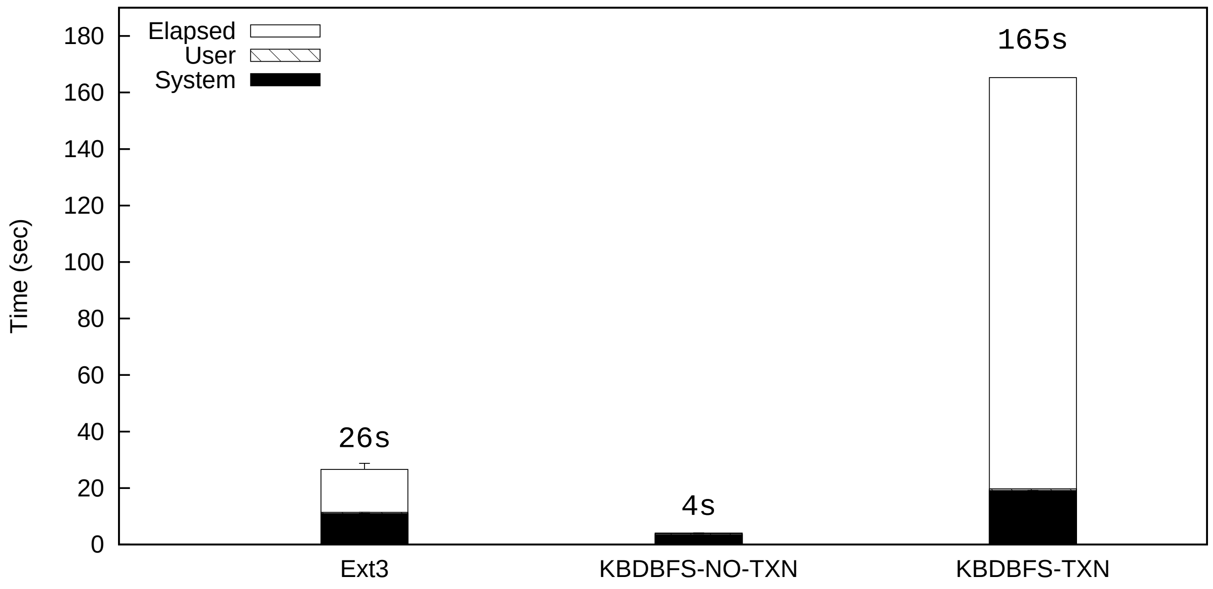

Figure 4.2: Postmark results

Postmark results

The Postmark results on KBDBFS with transactions disabled show an 84% gain in elapsed time over Ext3. Most of the gain is seen to be in I/O time, which can be explained by the overhead in journaling that Ext3 has over KBDBFS with transactions disabled. The results on KBDBFS with transactions enabled, on the other hand, show that KBDBFS-TXN is 6.3 times slower than Ext3. KBDBFS with transactions enabled must write all data and meta-data twice: once to the log file, and the second time to the database file itself. This is to rollback if a transaction is aborted. Ext3 only journals the meta-data, and writes the data directly in place to the device. This prevents Ext3 from rolling back data, but not writing to the journal considerably reduces I/O time for this benchmark.

Extended Attributes and ACLs We used a 2.4.24 kernel with version 0.8.68 of the standard EA patch and version 0.8.70 of the ACL patch [7]. We used version 2.4.12 of the `setfattr` and `getfattr` utilities. We used version 5.1.2 of `coreutils` with a patch applied for EA and ACL support.

The benchmarks for both EAs and ACLs were conducted in two modes: KBDBFS-TXN with transactions support, and KBDBFS-NO-TXN without transactions support. In the KBDBFS-TXN mode, the transactions are turned on for EAs and ACLs and in the KBDBFS-NO-TXN mode they are turned off. With transactions turned on, the transaction is committed as soon as the corresponding database operations complete successfully. When transactions are turned off, the EAs and ACLs are written to the database when the inode is written to the meta-data database.

We first discuss the microbenchmarks for EAs followed by the ACL benchmarks.

The microbenchmark for EAs consisted of the following steps. We created 100 files with `touch` in a directory. We used `setfattr` to set 100 EAs for each file. We then used `getfattr` to print the value of the attribute given its name. Afterward we listed all the EAs associated with each file. Finally, EAs were deleted with `setfattr`. Figure 4.3 shows the wait, user, and system time for the benchmark. The elapsed time overhead over Ext3 is 2.6% in KBDBFS-NO-TXN mode and 14.3% in KBDBFS-TXN mode. The overheads are mainly due to I/O with transactions being committed after every successful operation. We see a gain of 7% in CPU utilization with this benchmark. This can be attributed to the efficient traversals with B-tree against Ext3's linear search with linked lists. The overheads are small even under these worst-case conditions. For a common user workload, the overheads can be expected to be even smaller.

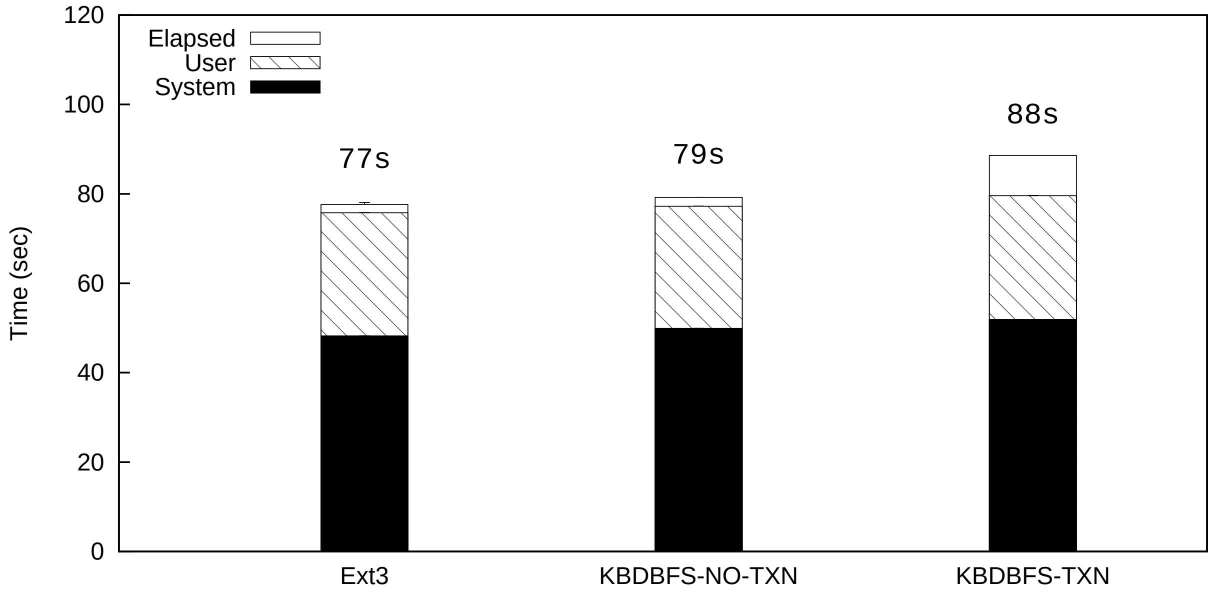

Figure 4.3: EA microbenchmark results comparing Ext3 with KBDBFS.

EA microbenchmark results comparing Ext3 with KBDBFS.

We also compared ACL performance on Ext3 and KBDBFS with two microbenchmarks. In the first benchmark, we created 5,000 files and associated five ACL entries with each file. `ls -l` calls the `getxattr` system call to fetch the ACLs for each file. If ACLs are associated with the file, `ls -l` displays an additional "+" after file permissions are displayed. We see an overhead of 2.1% without transactions and 2.4% with transactions over Ext3 using the `ls -l` command. There is a gain of 0.2% in CPU utilization with this benchmark. This is again explained by the efficient traversals with B-trees.

The second benchmark involved copying 5,000 files with ACLs. We used the `cp -p` command to copy the files while preserving the ACLs. This benchmark shows a gain of 8% in elapsed time and 14% in system time over Ext3 with no transactions enabled. This can be attributed to the journaling overhead of Ext3 over KBDBFS without transactions support. Enabling transactions in KBDBFS gives us an overhead of 84% over Ext3, but with the added reliability of the Berkeley Database. Transactions require writing to a log file and then flushing data to disk on a database `commit` operation, which explains why most of the overhead in this mode is in I/O time.

Chapter 5

Applications of KBDB

We have used the core KBDB component in other projects aside from KBDBFS. We believe that an embedded database, like KBDB, is an important reusable operating system component. We used KBDB in the following projects: a stackable file system that can add extended attribute and ACL support to any other file system, a persistent, registry-like `/proc` file system, data and meta-data integrity verification for an encryption file system called NCryptfs, an in-kernel intrusion detection file system, and adding file handle security features to NFS [8, 20, 21, 23]. Each project took our graduate students between one and eight person-weeks to develop.

EAFS: Extended Attribute File System Originally, the extended attributes and ACL support in KBDBFS was designed and built as a separate stackable file system, *EAFS*. EAFS can be layered on top of any other Linux file system to provide flexible and efficient extended attributes, and strong and flexible access control to legacy file systems.

Current file systems only support extended attributes and ACLs with arbitrary limitations on either the size of each attribute, the number of attributes per file, or the total size of all attributes for a file. The user is forced to deal with some combination of these limitations. With our stackable file system, we were able to extend extended attributes and ACL support to all Linux file systems, even those that did not support extended attributes and ACLs while keeping development, debugging and maintenance minimal unlike complex file system specific code. We provide ACL support even to file systems like FAT that traditionally do not implement a strong security model. We were able to provide the advantages of ACLs and enhance security on all these file systems uniformly, while keeping the implementation simple, addressing the current limitations and maintaining the existing Unix permission semantics.

Procdb The `/proc` file system on Linux can be used to obtain information about the system and to change various kernel parameters such as the maximum number of file handles that the Linux kernel will allocate at run-time. This functionality is similar to a *kernel registry*. However, one drawback of `/proc` is that it does not save the values of these kernel parameters in a persistent manner. A *persistent* and unified system registry lends itself to backups, sharing configurations, and centralizing all system configurations in a single place, thus making system cloning easier.

We added persistence to the `/proc` file system by introducing an underlying database layer. The modified `/proc` (Procdb) captures writes to `/proc` and updates a database that stores the current system configuration state as key-value pairs. We initialize this database when the system is booting with the existing values in `proc`. Storing state as key-value pairs suitably captures the logical structure of `/proc` entries.

File System Integrity We added data and meta-data integrity verification to the stackable encryption file system, NCryptfs, through checksums [23]. Our implementation of file system checksumming was primarily aimed at detecting malicious modifications of data and meta-data, especially if files are accessed over NFS.

Data on disk can be tampered with either by gaining access to the raw disk, effectively bypassing the file system, or by gaining physical access to the disk and modifying it on a different operating system. These modifications can be detected using checksumming. We implemented page-level checksumming for NCryptfs, and stored these checksums in a database. The database stored a tuple containing the inode number and page index of the page as the key that maps to the checksum for that page. Meta-data checksums were stored in a separate database, indexed by the inode number of the file. This checksum was computed over important fields of the inode object, such as `mtime`, `atime`, and `size`.

I^{3}FS We developed an on-access in-kernel intrusion detection file system using KBDB. Our system, called *I^{3}FS*, is an on-access integrity checking file system that compares the checksums of files in real-time [8]. It uses cryptographic checksums, stored in a KBDB database, to detect unauthorized modifications to files and performs necessary actions as configured. I^{3}FS is a stackable file system which can be mounted over any underlying file system (such as Ext3 or NFS). I^{3}FS also uses KBDB to store per-file and per-directory checksumming policies with inheritance.

NFS File Handle Security Each file on an NFS server is uniquely identified by a persistent file handle that is used whenever a client performs any NFS operation. NFS file handles reveal significant amounts of information about the server. If attackers can sniff the file handle, then they may be able to obtain useful information [21]. For example, the encoding used by a file handle indicates which operating system the server is running and when it was installed, allowing attackers to target their attacks to known vulnerabilities.

We modified an NFS server to persistently map file handles to random data to prevent information leaks and file handle guessing (this prevents users from bypassing normal logging facilities). We used the KBDB database module to store this mapping, as the mapping has to be persistent to avoid stale file handles in case the server is rebooted.

Chapter 6

Related Work

In this section we briefly survey past works that attempted to integrate databases and OSs together. These fall into three categories of relevance: file system interfaces to databases, file systems built using a database, and operating systems integrated with a database.

File System Interfaces to Databases The Inversion File System is a simple user-level wrapper library to access files stored in a database [14]. It provides a file-system–like interface to an existing database. Inversion is built on top of a DBMS; it takes advantage of low-level DBMS services to provide transaction protection, fine-grained time travel, and fast crash recovery for user files and file system meta-data. The current implementation of Inversion requires linking applications with a special library in order to access Inversion data. In contrast, our work provides a standard file system interface to all user applications without requiring them to be modified or relinked.

File Systems Built Using a Database Gehani et al. built a file system interface to the Ode object-oriented database, called OdeFS [5]. Database objects are accessed and manipulated like files in a traditional file system, using standard Unix commands. OdeFS is implemented as a user-level network file server, using the NFS protocol. OdeFS commands are translated into calls to the underlying Unix file system or the Ode object manager.

Oracle's Internet File System (iFS) presents a file system interface using Oracle as the backing store [15]. Aside from a user-level NFS server implementation that provides a traditional file system interface to applications, iFS also supports several additional network interfaces to the file system: AppleTalk, Samba, HTTP, and FTP. The iFS is a commercial product built on top of a heavy-weight DBMS; and it is also not freely available.

The Database File System (DBFS) is a block-structured file system developed on top of the Berkeley DB database and is similar to KBDBFS [13]. The Database File System is implemented as a user-level NFS file server, and uses the Berkeley Database as its backing store. To access data, DBFS provides a user-level library and a file system interface.

Operating Systems Integrated with Databases The WinFS file system is part of the upcoming *Longhorn* OS release from Microsoft [12]. Longhorn comes with a full-fledged SQL DBMS integrated into the OS. WinFS uses the database as a backing store for all files. WinFS reports having new features such as presenting data as a directed acyclic graph instead of a tree as in conventional file systems. The organization of data in this manner will enable more complex queries to search for data as opposed to the conventional method of storing data in a tree structure. Data organization, searching, and sharing are expected to be more versatile with WinFS. We chose to use an embedded, small-footprint database to minimize overheads; in comparison, a full-blown SQL database such as proposed by Longhorn, with query processing and all, may cause high overheads in the entire OS.

The Pick operating system was designed to process business data, such as inventory or sales leads [17]. Pick does not have a traditional file system, but rather uses a flat file name-space per user account, where each file contained named items. Pick essentially combines the operating system, query language, and database server into one system.

Chapter 7

Conclusions

We have designed and built an in-kernel file system (KBDBFS) built on top of an in-kernel database (KBDB). Using a database as the backing store significantly cut down on development time and makes the file system more extensible. In only 7.5 person-months, we have developed a file system that supports all basic POSIX operations, plus has support for retrieving all of a file's names, extended attributes, and ACLs. The efficient storage routines of the database give us acceptable performance for both user-like workloads.

We export some transaction-like functionality to user-level processes. A process can begin, abort, or commit a transaction that encompasses multiple file system operations. This allows applications to more reliably perform fundamental tasks like appending data to a file, listing the contents of a directory, or renaming multiple files.

Part of our investigation also centers around the question of "Does a database make sense as a reusable operating system component?" We believe the answer is yes. An embedded database such as BDB, certainly has a place within the operating system. The database itself is small, and does not place any undue burden on the kernel. Development times for applications persistently storing data are reduced and the resulting solution is reliable and performs well. When the database itself improves, all of the applications that make use of it do as well.

7.0.1 Future Work

The transaction model exported to user-space is relatively simplistic. Each process can begin, abort, or commit a single transaction. We plan to develop a more flexible system that allows processes to manage multiple transactions or to share transactions. Cross-file system transactions would allow more reliable storage than is presently available. Moving a file within the same file system results in a rename, but moving a file across file systems results in: (1) removing the destination, (2) copying the source to the destination, and (3) removing the source. This breaks a key property of the rename system call—that if the call fails, then the destination is unchanged. With cross-file system transactions, if the copy or unlink fail then the destination file would not be lost—the same properties that are guaranteed by rename.

We plan to develop a file system with complete ACID properties. Not only will the file system itself need to change, but the OS itself will need to be changed to allow the database to manage access to data. To provide isolation, all accesses will need to be served through the database. Pages and buffers, inodes, and directory name lookup caches all need to be controlled by KBDB. At the same time, the OS infrastructure needs to remain in place so that ACID-compliant file systems can coexist with traditional file systems. Because OS caches will be bypassed, the database cache itself will become more important. Therefore, like other OS caches, the database cache should be modified to adapt to resource availability: it should expand and contract as machine resources become available or scarce.

Bibliography

[1] C. Anderson. xFS Attribute Manager Design. Technical report, October 1993. `http://oss.sgi.com/projects/xfs/design_docs/xfsdocs93_pdf/attributes.pdf`.

[2] A. Aranya, C. P. Wright, and E. Zadok. Tracefs: A File System to Trace Them All. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004)*, pages 129–143, San Francisco, CA, March/April 2004.

[3] M. I. Bushnell. The HURD: Towards a new strategy of OS design. *GNU's Bulletin*, 1994. `www.gnu.org/software/hurd/hurd.html`.

[4] D. Ellard and M. Seltzer. NFS Tricks and Benchmarking Traps. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, pages 101–114, San Antonio, TX, June 2003.

[5] N. H. Gehani, H. V. Jagadish, and W. D. Roome. OdeFS: A File System Interface to an Object-Oriented Database. In *Proceedings of the Twentieth International Conference on Very Large Databases*, pages 249–260, Santiago, Chile, September 1994. Springer-Verlag Heidelberg.

[6] A. Grünbacher. POSIX Access Control Lists on Linux. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, pages 259–272, San Antonio, TX, June 2003.

[7] A. Grünbacher. Linux EA/ACL Downloads, 2004. `http://acl.bestbits.at`.

[8] A. Kashyap, S. Patil, G. Sivathanu, and E. Zadok. I3FS: An In-Kernel Integrity Checker and Intrusion Detection File System. In *Proceedings of the 18th USENIX Large Installation System Administration Conference (LISA 2004)*, pages 69–79, Atlanta, GA, November 2004.

[9] J. Katcher. PostMark: A New Filesystem Benchmark. Technical Report TR3022, Network Appliance, 1997. `www.netapp.com/tech_library/3022.html`.

[10] D. Kleikamp and S. Best. How the Journaled File System handles the on-disk layout, May 2000. `www-106.ibm.com/developerworks/library/l-jfslayout/`.

[11] J. Mahoney. ReiserFS Extended Attributes announcement, November 2002. `http://marc.theaimsgroup.com/?l=reiserfs&m=103765962910834&w=2`.

[12] Microsoft Corporation. Microsoft MSDN WinFS Documentation. `http://longhorn.msdn.microsoft.com/lhsdk/winfs/daovrwelcometowinfs.aspx`, 2003.

[13] N. Murphy, M. Tonkelowitz, and M. Vernal. The Design and Implementation of the Database File System. www.eecs.harvard.edu/~vernal/learn/cs261r/index.shtml, January 2002.

[14] M. A. Olson. The Design and Implementation of the Inversion File System. In *Proceedings of the Winter 1993 USENIX Technical Conference*, pages 205–217, San Diego, CA, January 1993. USENIX.

[15] Oracle Corporation. Oracle Internet File System Archive Documentation. http://otn.oracle.com/documentation/ifs_arch.html, October 2000.

[16] J. S. Pendry, N. Williams, and E. Zadok. *Am-utils User Manual*, 6.1b3 edition, July 2003. www.am-utils.org.

[17] *Encyclopedia Pick*. Pick Systems, third edition, December 1993.

[18] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *Proceedings of First USENIX conference on File and Storage Technologies*, pages 89–101, Monterey, CA, January 2002.

[19] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R.W. Carton, and J. Ofir. Deciding When to Forget in the Elephant File System. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 110–123, Charleston, SC, December 1999.

[20] G. Sivathanu, C. P. Wright, and E. Zadok. Enhancing File System Integrity Through Checksums. Technical Report FSL-04-04, Computer Science Department, Stony Brook University, May 2004. www.fsl.cs.sunysb.edu/docs/nc-checksum-tr/nc-checksum.pdf.

[21] A. Traeger, A. Rai, C. P. Wright, and E. Zadok. NFS File Handle Security. Technical Report FSL-04-03, Computer Science Department, Stony Brook University, May 2004. www.fsl.cs.sunysb.edu/docs/nfscrack-tr/nfscrack.pdf.

[22] F. von Leitner. diet libc – a libc optimized for small size. www.fefe.de/dietlibc, July 2004.

[23] C. P. Wright, M. Martino, and E. Zadok. NCryptfs: A Secure and Convenient Cryptographic File System. In *Proceedings of the Annual USENIX Technical Conference*, pages 197–210, San Antonio, TX, June 2003.

[24] E. Zadok and A. Dupuy. HLFSD: Delivering Email to your $HOME. In *Proceedings of the Seventh USENIX Systems Administration Conference (LISA VII)*, pages 243–254, Monterey, CA, November 1993.