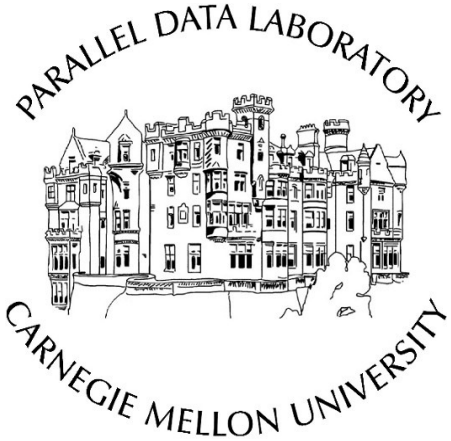


TABLEFS: Enhancing Metadata Efficiency in the Local File System

Kai Ren, Garth Gibson

CMU-PDL-13-102
REVISED VERSION OF CMU-PDL-12-110
January 2013

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Acknowledgements: This research is supported in part by The Gordon and Betty Moore Foundation, NSF under award, SCI-0430781 and CCF-1019104, Qatar National Research Foundation 09-1116-1-172, DOE/Los Alamos National Laboratory, under contract number DE-AC52-06NA25396/161465-1, by Intel as part of the Intel Science and Technology Center for Cloud Computing (ISTC-CC), by gifts from Yahoo!, APC, EMC, Facebook, Fusion-IO, Google, Hewlett-Packard, Hitachi, Huawei, IBM, Intel, Microsoft, NEC, NetApp, Oracle, Panasas, Riverbed, Samsung, Seagate, STEC, Symantec, and VMware. We thank the member companies of the PDL Consortium for their interest, insights, feedback, and support.

Keywords: TableFS, File System, File System Metadata, NoSQL Database, LSM Tree

Abstract

File systems that manage magnetic disks have long recognized the importance of sequential allocation and large transfer sizes for file data. Fast random access has dominated metadata lookup data structures with increasing use of B-trees on-disk. Yet our experiments with workloads dominated by metadata and small file access indicate that even sophisticated local disk file systems like Ext4, XFS and Btrfs leave a lot of opportunity for performance improvement in workloads dominated by metadata and small files.

In this paper we present a stacked file system, TABLEFS*, which uses another local file system as an object store.* TABLEFS *organizes all metadata into a single sparse table backed on disk using a Log-Structured Merge (LSM) tree, LevelDB in our experiments. By stacking,* TABLEFS *asks only for efficient large file allocation and access from the local file system. By using an LSM tree,* TABLEFS *ensures metadata is written to disk in large, non-overwrite, sorted and indexed logs. Even an inefficient FUSE based user level implementation of* TABLEFS *can perform comparably to Ext4, XFS and Btrfs on data-intensive benchmarks, and can outperform them by 50% to as much as 1000% for metadata-intensive workloads. Such promising performance results from* TABLEFS *suggest that local disk file systems can be significantly improved by more aggressive aggregation and batching of metadata updates.*

1 Introduction

In the last decade parallel and Internet service file systems have demonstrated effective scaling for high bandwidth, large file transfers [13, 17, 26, 39, 40, 49]. The same, however, is not true for workloads that are dominated by metadata and tiny file access [35, 50]. Instead there has emerged a class of scalable small-data storage systems, commonly called key-value stores, that emphasize simple (NoSQL) interfaces and large in-memory caches [2, 24, 34].

Some of these key-value stores feature high rates of change and efficient out-of-memory Log-structured Merge (LSM) tree structures [8, 23, 33]. An LSM tree can provide fast random updates, inserts and deletes without scarificing lookup performance [5]. We believe that file systems should adopt LSM tree techniques used by modern key-value stores to represent metadata and tiny files, because LSM trees aggressively aggregate metadata. Moreover, today's key-value store implementations are "thin" enough to provide the performance levels required by file systems.

In this paper we present experiments in the most mature and restrictive of environments: a local file system managing one magnetic hard disk. We used a LevelDB key-value store [23] to implement TABLEFS, our POSIX-compliant stacked file system, which represents metadata and tiny files as key-value pairs. Our results show that for workloads dominated by metadata and tiny files, it is possible to improve the performance of the most modern local file systems in Linux by as much as an order of magnitude. Our demonstration is more compelling because it begins disadvantaged: we use an interposed file system layer [1, 52] that represents metadata and tiny files in a LevelDB store whose LSM tree and log segments are stored in the same local file systems we compete with.

2 Background

Even in the era of big data, most things in many file systems are small [10, 29]. Inevitably, scalable systems should expect the numbers of small files to soon achieve and exceed billions, a known challenge for both the largest [35] and most local file systems [50]. In this section we review implementation details of the systems employed in our experiments: Ext4, XFS, Btrfs and LevelDB.

2.1 Local File System Structures

Ext4[27] is the fourth generation of Linux ext file systems, and, of the three we study, the most like traditional UNIX file systems. Ext4 divides the disk into block groups, similar to cylinder groups in traditional UNIX, and stores in each block group a copy of the superblock, a block group descriptor, a bitmap describing free data blocks, a table of inodes and a bitmap describing free inodes, in addition to the actual data blocks. Inodes contain a file's attributes (such as the file's inode number, ownership, access mode, file size and timestamps) and four extent pointers for data extents or a tree of data extents. The inode of a directory contains links to a HTree (similar to B-Tree) that can be one or two levels deep, based on a 32 bit hash of the directory entry's name. By default only changes to metadata are journaled for durability, and Ext4 asynchronously commits its journal to disk every five seconds. When committing pending data and metadata, data blocks are written to disk before the associated metadata is written to disk.

XFS[48], originally developed by SGI, aggressively and pervasively uses B+ trees to manage all file structures: free space maps, file extent maps, directory entry indices and dynamically allocated inodes. Because all file sizes, disk addresses and inode numbers are 64 bits in XFS, index structures can be large. To reduce the size of these structures XFS partitions the disk into allocation groups, clusters allocation in an allocation group and uses allocation group relative pointers. Free extents are represented in two B+ trees: one indexed by the starting address of the extent and the other indexed by the length of the extent, to enable efficient search for an appropriately sized extent. Inodes contain either a direct extent map, or a B+

tree of extent maps. Each allocation group has a B+ tree indexed by inode number. Inodes for directories have a B+ tree for directory entries, indexed by a 32 bit hash of the entry's file name. XFS also journals metadata for durability, committing the journal asynchronously when a log buffer (256 KB by default) fills or synchronously on request.

Btrfs[22, 37] is the newest and most sophisticated local file system in our comparison set. Inspired by Rodeh's copy-on-write B-tree[36], as well as features of XFS, NetApp's WAFL and Sun's ZFS[3, 18], Btrfs copies any B-tree node to an unallocated location when it is modified. Provided the modified nodes can be allocated contiguously, B-tree update writing can be highly sequential; however more data must be written than is minimally needed (write amplification). The other significant feature of Btrfs is its collocation of different metadata components in the same B-tree, called the FS tree. The FS tree is indexed by (inode number, type, offset) and it contains inodes, directory entries and file extent maps, distinguished by a type field: INODE_ITEM for inodes, DIR_ITEM and DIR_INDEX for directory entries, and EXTENT_DATA_REF for file extent maps. Directory entries are stored twice so that they can be ordered differently: in one the offset field of the FS tree index (for the directory's inode) is the hash of the entry's name, for fast single entry lookup, and in the other the offset field is the child file's inode number. The latter allows a range scan of the FS tree to list the inodes of child files and accelerate user operations such as $ls + stat$. Btrfs, by default, delays writes for 30 seconds to increase disk efficiency, and metadata and data are in the same delay queue.

2.2 LevelDB and its LSM Tree

Inspired by a simpler structure in BigTable[8], LevelDB [23] is an open-source key-value storage library that features an Log-Structured Merge (LSM) tree [33] for on-disk storage. It provides simple APIs such as GET, PUT, DELETE and SCAN (an iterator). Unlike BigTable, not even single row transactions are supported in LevelDB. Because TABLEFS uses LevelDB, we will review its design in greater detail in this section.

In a simple understanding of an LSM tree, an memory buffer cache delays writing new and changed entries until it has a significant amount of changes to record on disk. Delay writes are made more durable by redundantly recording new and changed entries in a write-ahead log, which is pushed to disk periodically and asynchronously by default.

In LevelDB, by default, a set of changes are spilled to disk when the total size of modified entries exceeds 4 MB. When a spill is triggered, called a minor compaction, the changed entries are sorted, indexed and written to disk in a format known as SSTable[8]. These entries may then be discarded by the memory buffer and can be reloaded by searching each SSTable on disk, possibly stopping when the first match occurs if the SSTables are searched from most recent to oldest. The number of SSTables that need to be searched can be reduced by maintaining a Bloom filter[7] on each, but with increasing numbers of records the disk access cost of finding a record not in memory increases. Scan operations in LevelDB are used to find neighbor entries, or to iterate through all key-value pairs within a range. When performing a scan operation, LevelDB first searches each SSTable to place a cursor; it then increments cursors in the multiple SSTables and merges key-value pairs in sorted order. Major compaction, or simply "compaction", is the process of combining multiple SSTables into a smaller number of SSTables by merge sort. Compaction is similar to *online defragmentation* in traditional file systems and *cleaning* in LFS [38].

As illustrated in Figure 1, LevelDB extends this simple approach to further reduce read costs by dividing SSTables into sets, or levels. Levels are numbered starting from 0, and levels with a smaller number are referenced as "lower" levels. The 0th level of SSTables follows a simple formulation: each SSTable in this level may contain entries with any key/value, based on what was in memory at the time of its spill. LevelDB's SSTables in level $L > 0$ are the results of compacting SSTables from level L or $L-1$. In these higher levels, LevelDB maintains the following invariant: the key range spanning each SSTable is disjoint from the key range of all other SSTables at that level and each SSTable is limited in size (2MB by default).

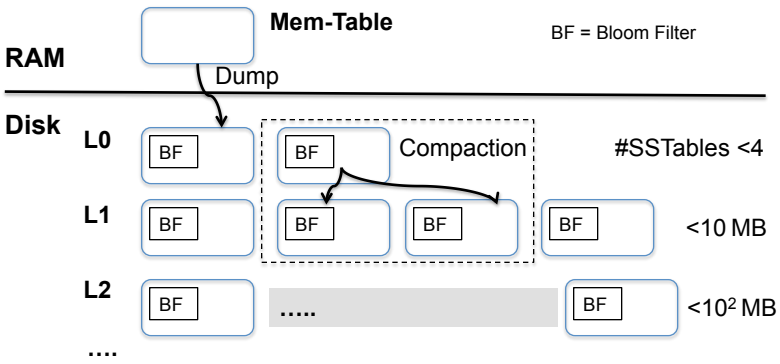


Figure 1: *LevelDB represents data on disk in multiple SSTables that store sorted key-value pairs. SSTables are grouped into different levels with lower-numbered levels containing more recently inserted key-value pairs. Finding a specific pair on disk may search up to all SSTables in level 0 and at most one in each higher-numbered level. Compaction is the process of combining SSTables by merge sort into higher-numbered levels.*

Therefore querying for an entry in the higher levels only need to read at most one SSTable in each level. LevelDB also sizes each level differentially: all SSTables have the same maximum size and the sum of the sizes of all SSTables at level L will not exceed 10^L MB. This ensures that the number of levels, that is, the maximum number of SSTables that need to be searched in the higher levels, grows logarithmically with increasing numbers of entries.

When LevelDB decides to compact an SSTable at level L, it picks one, finds all other SSTables at the same level and level $L+1$ that have an overlapping key range, and then merge sorts all of these SSTables, producing a set of SSTables with disjoint ranges at the next higher level. If an SSTable at level 0 is selected, it is not unlikely that many or all other SSTables at level 0 will also be compacted, and many SSTables at level 1 may be included. But at higher levels most compactions will involve a smaller number of SSTables. To select when and what to compact there is a weight associated with compacting each SSTable, and the number of SSTables at level 0 is held in check (by default compaction will be triggered if there are more than four SSTables at level 0). There are also counts associated with SSTables that are searched when looking for an entry, and hotter SSTables will be compacted sooner. Finally, only one compaction runs at a time.

3 TABLEFS

As shown in Figure 2(a), TABLEFS exploits the FUSE user level file system infrastructure to interpose on top of the local file system. TABLEFS represents directories, inodes and small files in one all encompassing table, and only writes to the local disk large objects such as write-ahead logs, SSTables, and files whose size is large.

3.1 Local File System as Object Store

There is no explicit space management in TABLEFS. Instead, it uses the local file system for allocation and storage of objects. Because TABLEFS packs directories, inodes and small files into a LevelDB table, and LevelDB stores sorted logs (SSTables) of about 2MB each, the local file system sees many fewer, larger objects. We use Ext4 as the object store for TABLEFS in all experiments.

Files larger than T bytes are stored directly in the object store named according to their inode number. The object store uses a two-level directory tree in the local file system, storing a file with inode number I as "/LargeFileStore/J/I" where $J = I \div 10000$. This is to circumvent any scalability limits on directory entries in the underlying local file systems. In TABLEFS today, T, the threshold for blobbing a file is 4KB, which

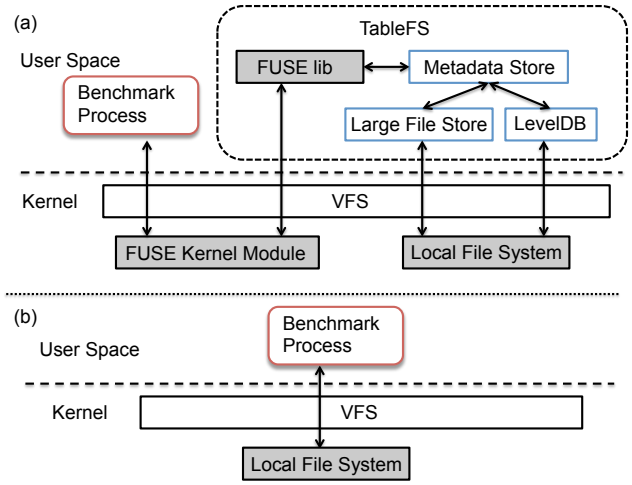


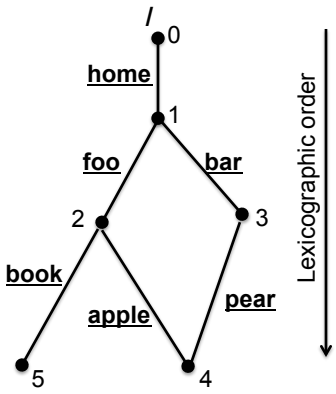
Figure 2: *(a) The architecture of* TABLEFS. *A FUSE kernel module redirects file system calls from a benchmark process to* TABLEFS*, and* TABLEFS *stores objects into either LevelDB or a large file store. (b) When we benchmark a local file system, there is no FUSE overhead to be paid.*

is the median size of files in desktop workloads [29], although others have suggested T be at least 256KB and perhaps as large as 1MB [42].

3.2 Table Schema

TABLEFS's metadata store aggregates directory entries, inode attributes and small files into one LevelDB table with a row for each file. To link together the hierarchical structure of the user's namespace, the rows of the table are ordered by a 128-bit key consisting of the 64-bit inode number of a file's parent directory and a 64-bit hash value of its filename string (final component of its pathname). The value of a row contains the file's full name and inode attributes, such as inode number, ownership, access mode, file size and timestamps (*struct stat* in Linux). For small files, the file's row also contains the file's data.

Figure 3 shows an example of storing a sample file system's metadata into one LevelDB table.



Key	Value
<0,hash(home)>	1, "home", struct *stat*
<1,hash(foo)>	2, "foo", struct *stat*
<1,hash(bar)>	3, "bar", struct *stat*
<2,hash(apple)>	4, "apple", hard link
<2,hash(book)>	5, "book", struct *stat*, inline small file(<4KB)
<3,hash(pear)>	4, "pear", hard link
<4,null>	4, struct *stat*, large file pointer (> 4KB)

Figure 3: *An example illustrates table schema used by* TABLEFS*'s metadata store. The file with inode number 4 has two hard links, one called "apple" from directory foo and the other called "pear" from directory bar.*

All entries in the same directory have rows that share the same first 64 bits of their table key. For *readdir* operations, once the inode number of the target directory has been retrieved, a scan sequentially lists all entries having the directory's inode number as the first 64 bits of their table key. To resolve a

single pathname, TABLEFS starts searching from the root inode, which has a well-known inode number (0). Traversing the user's directory tree involves constructing a search key by concatenating the inode number of current directory with the hash of next component name in the pathname. Unlike Btrfs, TABLEFS does not need the second version of each directory entry because the entire attributes are returned in the *readdir* scan.

3.3 Hard Links

Hard links, as usual, are a special case because two or more rows must have the same inode attributes and data. Whenever TABLEFS creates the second hard link to a file, it creates a separate row for the file itself, with a null name, and its own inode number as its parent's inode number in the row key. As illustrated in Figure 3, creating a hard link also modifies the directory entry such that each row naming the file has an attribute indicating the directory entry is a hard link to the file object's inode row.

3.4 Scan Operation Optimization

TABLEFS utilizes the scan operation provided by LevelDB to implement *readdir()* system call. The scan operation in LevelDB is designed to support iteration over arbitrary key ranges, which may require searching SSTables at each level. In such a case, Bloom filters cannot help to reduce the number of SSTables to search. However, in TABLEFS, *readdir()* only scans keys sharing the common prefix — the inode number of the searched directory. For each SSTable, an additional Bloom filter can be maintained, to keep track of all inode numbers that appear as the first 64 bit of row keys in the SSTable. Before starting an iterator in an SSTable for *readdir()*, TABLEFS can first check its Bloom filter to find out whether it contains any of the desired directory entries. Therefore, unnecessary iterations over SSTables that do not contain any of the requested directory entries can be avoided.

3.5 Inode Number Allocation

TABLEFS uses a global counter for allocating inode numbers. The counter increments when creating a new file or a new directory. Since we use 64-bit inode numbers, it will not soon be necessary to recycle the inode number of deleted entries. Coping with operating systems that use 32 bit inode numbers may require frequent inode number recycling, a problem beyond the scope of this paper and addressed by many file systems.

3.6 Locking and Consistency

LevelDB provides atomic insertion of a batch of writes but does not support atomic row read-modify-write operations. The atomic batch write guarantees that a sequence of updates to the database are applied in order, and committed to the write-ahead log atomically. Thus the *rename* operation can be implemented as a batch of two operations: insert the new directory entry and delete the stale entry. But for operations like *chmod* and *utime*, since all of an inode's attributes are stored in a single key-value pair, TABLEFS must read-modify-write attributes atomically. We implemented a light-weight locking mechanism in the TABLEFS core layer to ensure correctness under concurrent access.

3.7 Journaling

TABLEFS relies on LevelDB and the local file system to achieve journaling. LevelDB has its own write-ahead log that journals all updates to the table. LevelDB can be set to commit the log to disk synchronously

or asynchronously. To achieve a consistency guarantee similar to "ordered mode" in Ext4, TABLEFS forces LevelDB to commit the write-ahead log to disk periodically (by default it is committed every 5 seconds).

3.8 TABLEFS in the Kernel

A kernel-native TABLEFS file system is a stacked file system, similar to eCryptfs [14, 52], treating a second local file system as an object store, as shown in Figure 4(a). An implementation of a Log-Structured Merge (LSM) tree [33] used for storing TABLEFS in the associated object store, such as LevelDB [23], is likely to have an asynchronous compaction thread that is more conveniently executed at user level in a TABLEFS daemon, as illustrated in Figure 4(b).

For the experiments in this paper, we bracket the performance of a kernel-native TABLEFS (Figure 4(a)), between a FUSE-based user-level TABLEFS (Figure 4(b)) with no TABLEFS function in the kernel and all of TABLEFS in the user level FUSE daemon) and an application-embedded TABLEFS library, illustrated in Figure 4(c).

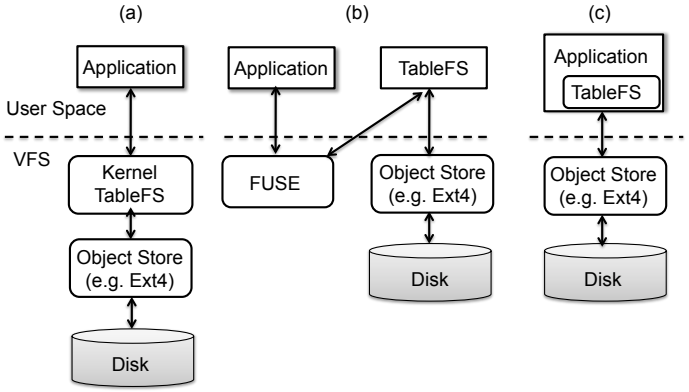


Figure 4: *Three different implementations of* TABLEFS*: (a) the kernel-native* TABLEFS*, (b) the FUSE version of* TABLEFS*, and (c) the library version of* TABLEFS*. In the following evaluation section, (b) and (c) are presented to bracket the performance of (a), which was not implemented.*

TABLEFS entirely at user-level in a FUSE daemon is unfairly slow because of the excess kernel crossings and scheduling delays experienced by FUSE file systems [6, 46]. TABLEFS embedded entirely in the benchmark application as a library is not sharable, and unrealistically fast because of the infrequency of system calls. We approximate the performance of a kernel-native TABLEFS using the library version and preceding each reference to the TABLEFS library with a write("*/dev/null*", *N* bytes) to account for the system call and data transfer overhead. *N* is chosen to match the size of data passed through each system call. More details on these models will be discussed in Section 4.3.

4 Evaluation

4.1 Evaluation System

We evaluate our TABLEFS prototype on Linux desktop computers equipped as follows:

Linux	Ubuntu 12.10, Kernel 3.6.6 64-bit version
CPU	AMD Opteron Processor 242 Dual Core
DRAM	16GB DDR SDRAM
Hard Disk	Western Digital WD2001FASS-00U0B0
	SATA, 7200rpm, 2TB
	Random Seeks 100 seeks/sec peak
	Sequential Reads 137.6 MB/sec peak
	Sequential Writes 135.4 MB/sec peak

We compare TABLEFS with Linux's most sophisticated local file systems: Ext4, XFS, and Btrfs. Ext4 is mounted with "ordered" journaling to force all data to be flushed out to disk before its metadata is committed to disk. By default, Ext4's journal is asynchronously committed to disks every 5 seconds. XFS and Btrfs use similar policies to asynchronously update journals. Btrfs, by default, duplicates metadata and calculates checksums for data and metadata. We disable both features (unavailable in the other file systems) when benchmarking Btrfs to avoid penalizing it. Since the tested filesystems have different inode sizes (Ext4 and XFS use 256 bytes and Btrfs uses 136 bytes), we pessimistically penalize TABLEFS by padding its inode attributes to 256 bytes. This slows down TABLEFS doing metadata-intensive workloads significantly, but it still performs quite well.

4.2 Data-Intensive Macrobenchmark

We run two sets of macrobenchmarks on the FUSE version of TABLEFS, which provides a full featured, transparent application service. Instead of using a metadata-intensive workload, emphasized in the previous and later sections of this paper, we emphasize data-intensive work in this section. Our goal is to demonstrate that TABLEFS is capable of reasonable performance for the traditional workloads that are often used to test local file systems.

Kernel build is a macrobenchmark that uses a Linux kernel compilation and related operations to compare TABLEFS's performance to the other tested file systems. In the kernel build test, we use the Linux 3.0.1 source tree (whose compressed tar archive is about 73 MB in size). In this test, we run four operations in this order:

- untar: Untar the source tarball;
- grep: Grep "nonexistent pattern" over all of the source tree;
- make: Run *make* inside the source tree;
- gzip: Gzip the entire source tree.

After compilation, the source tree contains 45,567 files with a total size of 551MB.

Table 1 shows the average runtime of three runs of these four macro-benchmarks using Ext4, XFS, Btrfs and TABLEFS-FUSE. Summing the operations, TABLEFS-FUSE is about 20% slower, but it is also paying significant overhead caused by moving all data through the user-level FUSE daemon and the kernel twice, instead of only through the kernel once, as illustrated in Figure 4. Table 1 also shows that the degraded performance of Ext4, XFS, and Btrfs when they are accessed through FUSE is about the same as TABLEFS-FUSE.

Postmark was designed to measure the performance of a file system used for e-mail, and web based services [20]. It creates a large number of small randomly-sized files between 512B and 4KB, performs a specified number of transactions on them, and then deletes all of them. Each transaction consists of two sub-transactions, with one being a create or delete and the other being a read or append. The configuration used

for these experiments consists of two million transactions on one million files, and the biases for transaction types are equal. The experiments were run against TABLEFS-FUSE with the available memory set to be 1400 MB, too small to fit the entire datasets in memory.

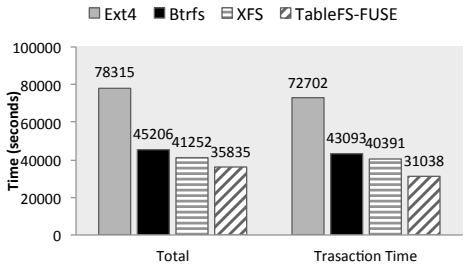


Figure 5: *The elapsed time for both the entire run of Postmark and the transactions phase of Postmark for the four tested file systems.*

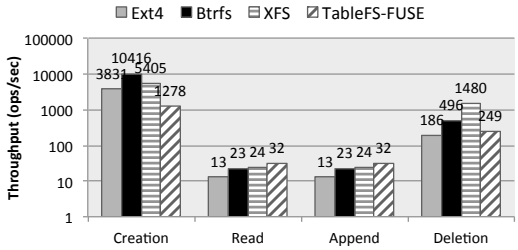


Figure 6: *Average throughput of each type of operation in Postmark benchmark.*

Figure 5 shows the Postmark results for the four tested file systems. TABLEFS outperforms other tested file systems by at least 23% during the transctions phase. Figure 6 gives the average throughput of each type of operations individually. TABLEFS runs faster than the other tested filesystems for *read* and *append*, and runs slower for the *creation* and *deletion*. In LevelDB, *deletion* is implemented by inserting entries with a deletion flag. The actual deletion is delayed until compaction procedure reclaims the deleted entries. Such an implementation is not as efficient as XFS and Ext4 for Postmark workloads, because XFS and Ext4 can reclaim deleted inodes whose inode numbers are continuous in a range more efficiently.

	Untar	Grep	Make	Gzip
Ext4	45	11	10132	1001
Btrfs	44	12	10187	996
XFS	44	11	10149	999
TABLEFS-FUSE	53	16	12083	1062
Ext4+FUSE	53	13	12107	1075
Btrfs+FUSE	52	14	12194	1066
XFS+FUSE	52	13	12163	1068

Table 1: *The elapsed time in seconds for unpacking, searching building and compressing the Linux 3.0.1 kernel package.*

4.3 TABLEFS-FUSE Overhead Analysis

To understand the overhead of FUSE in TABLEFS-FUSE, and estimate the performance of an in-kernel TABLEFS, we ran a micro-benchmark against TABLEFS-FUSE and TABLEFS-Library ((b) and (c) in Figure 4). This micro-benchmark creates one million zero-length files in one directory starting with an empty file system. The amount of memory available to the evaluation system is 1400 MB, almost enough to fit the benchmark in memory.

Figure 7 shows the total runtime of the experiment. TABLEFS-FUSE is about 3 times slower than TABLEFS-Libary.

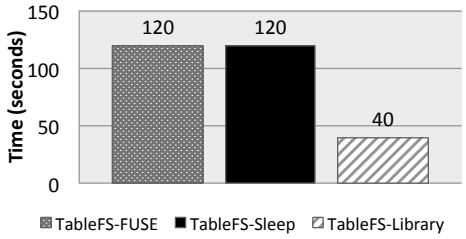


Figure 7: *The elapsed time for creating 1M zero-length files on three versions of* TABLEFS *(See Figure 4)*

.

Figure 8 shows the total disk traffic gathered from the Linux proc file system (*/proc/diskstats*) during the test. Relative to TABLEFS-Library, TABLEFS-FUSE writes almost as twice as many bytes to the disk, and reads almost 100 times as much. This additional disk traffic results from two sources: 1) under a slower insertion rate, LevelDB tends to compact more often; and 2) the FUSE framework populates the kernel's cache with its own version of inodes, competing with the local file system for cache memory.

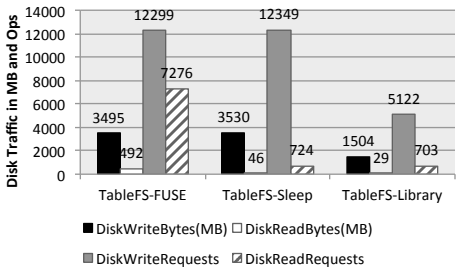


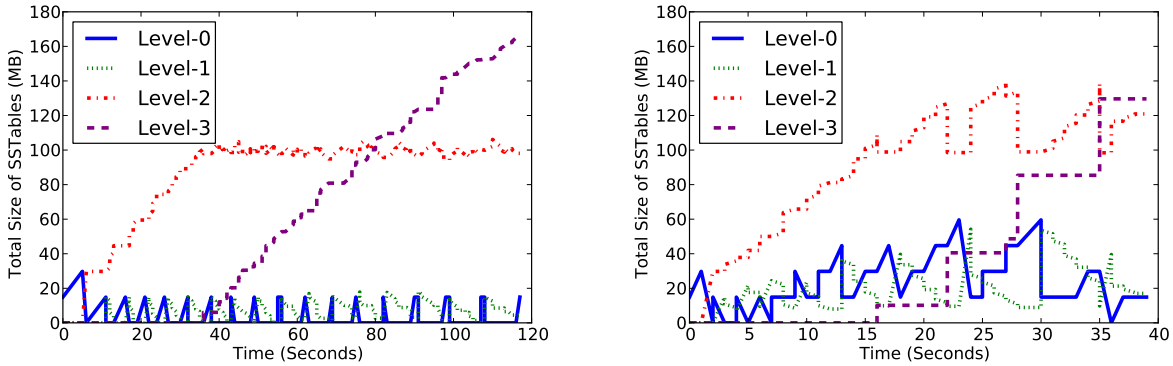
Figure 8: *Total disk traffic associated with Figure 7*

To illustrate the first point, we show LevelDB's compaction process during this test in Figure 9. Figure 9 shows the total size of SSTables in each Level over time. The compaction process will move SSTables from one level to the next level. For each compaction in Level 0, LevelDB will compact all SSTables with overlapping ranges (which in this benchmark will be almost all SSTables in level 0 and 1). At the end of a compaction, the next compaction will repeat similar work, except the number of level 0 SSTables will be proportional to the data insertion rate. When the insertion rate is slower (Figure 9(a)), compaction in Level 0 finds fewer overlapping SSTables than TABLEFS-Library (Figure 9(b)) in each compaction. In Figure 9(b), the level 0 size (blue line) exceeds 20MB for much of the test, while in 9(a) it never exceeds 20MB after the first compaction. Therefore, LevelDB does more compactions to integrate the same arriving log of changes when insertion is slower.

To negate the different compaction work, we deliberately slow down TABLEFS-Library to run at the same speed as TABLEFS-FUSE by adding *sleep 80ms* every 1000 operations (80ms was empirically derived

to match the run time of TABLEFS-FUSE). This model of TABLEFS is called TABLEFS-Sleep and is shown in Figure 8 and 9 (c). TABLEFS-Sleep causes almost the same pattern of compactions as does TABLEFS-FUSE and induces about the same write traffic (Figure 8). But unlike TABLEFS-FUSE, TABLEFS-Sleep can use more of the kernel page cache to store SSTables than TABLEFS-FUSE. Thus, as shown in Figure 8, TABLEFS-Sleep writes the same amount of data as TABLEFS-FUSE but does much less disk reading.

To estimate TABLEFS performance without FUSE overhead, we use TABLEFS-Library to avoid double caching and perform a *write*("*/dev/null*", *N* bytes) on every TABLEFS invocation to model the kernel's system call and argument data transfer overhead. This model is called TABLEFS-Predict and is used in the following sections to predict metadata efficiency of a kernel TABLEFS.



(a) TABLEFS-FUSE

(b) TABLEFS-Library

(c) TABLEFS-Sleep

Figure 9: *Changes of total size of SSTables in each level over time during the creation of 1M zero-length files for three* TABLEFS *models.* TABLEFS-*Sleep illustrates similar compaction behavior as does* TABLEFS-*FUSE.*

4.4 Metadata-Intensive Microbenchmark

Metadata-only Benchmark

In this section, we run four micro-benchmarks of the efficiency of pure metadata operations. Each micro-benchmark consists of two phases: a) create and b) test. For all four tests, the create phase is the same:

- *a) *create*: In "create", the benchmark application generates directories in depth first order, and then creates one million files in the appropriate parent directories in a random order, according to a realistic or synthesized namespace.

The test phase in the benchmark are:

- 1b) *null*: In test 1, the test phase is null because create is what we are measuring.
- 2b) *query*: This workload issues one million read or write queries to random (uniform) files or directories. A read query calls *stat* on the file, and a write query randomly does either a *chmod* or *utime* to update the mode or the timestamp attributes.
- 3b) *rename*: This workload issues a half million rename operations to random (uniform) files, moving the file to another randomly chosen directory.
- 4b) *delete*: This workload issues a half million delete operations to randomly chosen files.

The captured file system namespace used in the experiment was taken from one author's personal Ubuntu desktop. There were 172,252 directories, each with 11 files on average, and the average depth of the namespace is 8 directories. We also used the Impressions tool [4] to generate a "standard namespace". This synthetic namespace yields similar results, so its data is omitted from this paper. Between the create and test phase of each run, we umount and re-mount local filesystems to clear kernel caches. To test out-of-RAM performance, we limit the machine's available memory to 350MB which does not fit the entire test in memory. All tests were run for three times, and the coefficient of variation is less than 1%.

Figure 10 shows the test results averaged over three runs. The create phase of all tests had the same performance so we show it only once. For the other tests, we show only the second, test phase. Both TABLEFS-Predict and TABLEFS-FUSE runs are almost 2 to 3 times faster than the other local file systems in all tests.

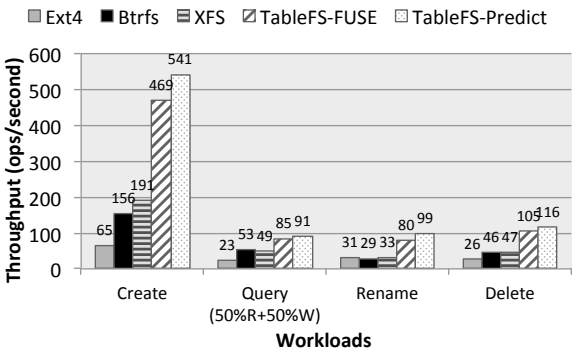


Figure 10: *Average throughput during four different workloads for five tested systems.*

Figure 11 shows the total number of disk read and write requests during the query workload, the test in which TABLEFS has the least advantage. Both versions of TABLEFS issue many fewer disk writes, effectively aggregating changes into larger sequential writes. For read requests, because of bloom filtering and in-memory indexing, TABLEFS issues fewer read requests. Therefore TABLEFS's total number of disk requests is smaller than the other tested file systems.

Scan Queries

In addition to point queries such as *stat, chmod* and *utime*, range queries such as *readdir* are important metadata operations. To test the performance of *readdir*, we modify the micro-benchmark to perform multiple *readdir* operations in the generated directory tree. To show the tradeoffs involved in embedding small files, we create 1KB files (with random data) instead of zero byte files. For the test phase, we use the following three operations:

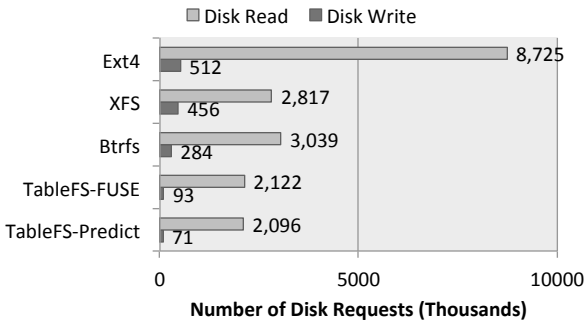


Figure 11: *Total number of disk read/write requests during 50%Read+50%Write query workload for five tested systems.*

- 5b) *readdir*: The benchmark application performs *readdir()* on 100,000 randomly picked directories.
- 6b) *readdir+stat*: The benchmark application performs *readdir()* on 100,000 randomly picked directories, and for each returned directory entry, performs a *stat* operation. This simulates "ls -l".
- 7b) *readdir+read*: Similar to *readdir+stat*, but for each returned directory entry, it reads the entire file (if returned entry is a file) instead of *stat*.

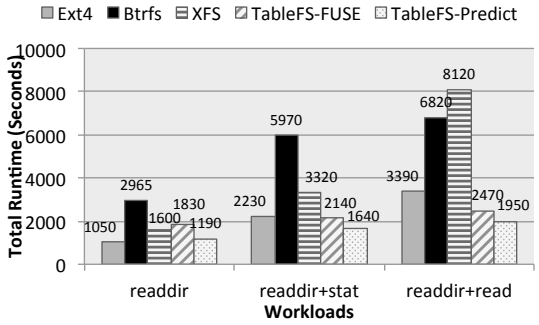


Figure 12: *Total run-time of three readdir workloads for five tested file systems.*

Figure 12 shows the total time needed to complete each *readdir* workload (the average of three runs). In the pure *readdir* workload, TABLEFS-Predict is slower than Ext4 because of read amplification, that is, for each *readdir* operation, TABLEFS fetches directory entries along with unnecessary inode attributes and file data. However, in the other two workloads when at least one of the attributes or file data is needed, TABLEFS is faster than Ext4, XFS, and Btrfs, since many random disk accesses are avoided by embedding inodes and small files.

Benchmark with Larger Directories

Because the scalability of small files is of topical interest [50], we modified the zero-byte file create phase to create 100 million files (a number of files rarely seen in the local file system today). In this benchmark, we allowed the memory available to the evaluation system to be the full 16GB of physical memory.

Figure 13 shows a timeline of the creation rate for four file systems. In the beginning of this test, there is a throughput spike that is caused by everything fitting in the cache. Later in the test, the creation rate of all tested file systems slows down because the non-existence test in each create is applied to ever larger

on-disk data structures. Btrfs suffers the most serious drop, slowing down to 100 operations per second at some points.

TABLEFS-FUSE maintains a more steady performance with an average speed of more than 2,200 operations per second and *is 10 times faster than all other tested file systems.*

All tested file systems have throughput fluctuations during the test and the behavior of TABLEFS's throughput is the smoothest than others. This kind of fluctuation might be caused by on disk data structure maintenance. In TABLEFS, this behavior is caused by compactions in LevelDB, in which SSTables are merged and sequentially written back to disk.

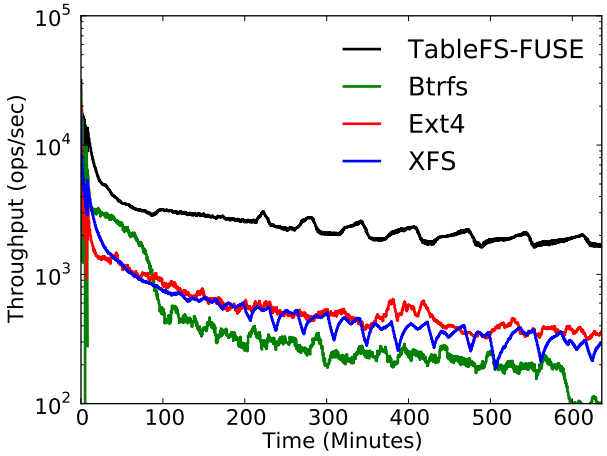


Figure 13: *Throughput during the first 650 minutes while creating 100 million zero-length files. We only graph the time until* TABLEFS-*FUSE finished inserting all 100 million zero-length files, because the other file systems were much slower.* TABLEFS-*FUSE is almost 10X faster than the other tested file systems in the later stage of this experiment. The data is sampled in every 10 seconds and smoothed over 100 seconds. The veritical axis is shown on a log scale.*

Solid State Drive Results

TABLEFS reduces disk seeks, so you might expect it to have less benefit on solid state drives, and you'd be right. We applied the "create-query" microbenchmark to a 120GB SATA II 2.5in Intel 520 Solid State Drive (SSD). Random read throughput is 15,000 IO/s at peak, and random write throughput peaks at 3,500 IO/s. Sequential read throughput peaks at 245MB/sec, and sequential write throughput peaks at 107MB/sec. Btrfs has a "ssd" optimization mount option which we enabled.

Figure 14 shows performance averaged over three runs of the create and query phases. In comparison to Figure 10, all results are about 10 times faster. Although TABLEFS is not the fastest, although TABLEFS-Predict is comparable to the fastest. Figure 15 shows the total number of disk requests and disk bytes moved during the query phase. Although TABLEFS achieves fewer disk writes, it reads much more data from SSD than XFS and Btfs. For use with solid state disks, LevelDB should be optimized differently, perhaps using SILT-like indexing [24], or VT-Tree compaction stitching [46].

5 Related Work

File system metadata is structured data, a natural fit for relational database techniques. However, because of their large size, complexity and slow speed, file system developers have long been reluctant to incorporate traditional databases into the lower levels of file systems [32, 47]. Modern stacked file systems often expand

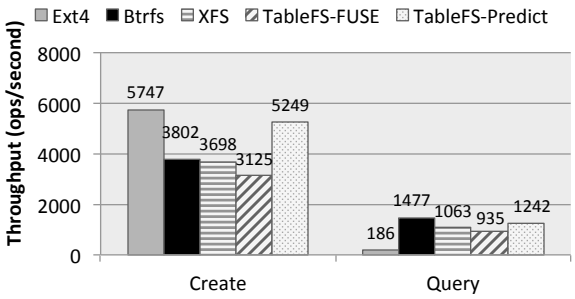
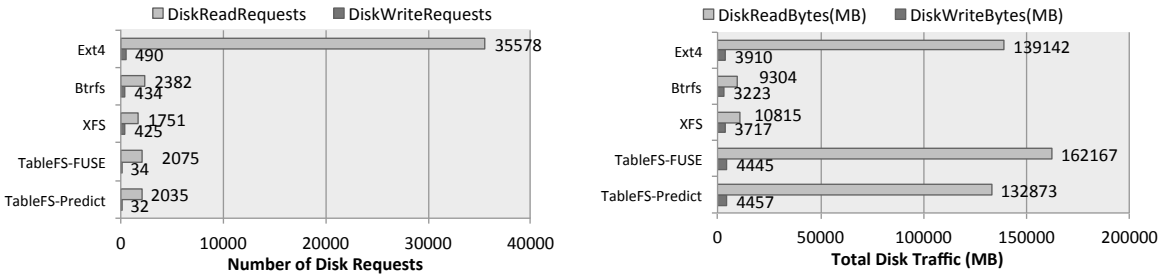


Figure 14: *Average throughput in the create and query workloads on an Intel 520 SSD for five tested file systems.*



(a) Disk Requests

(b) Disk Bytes

Figure 15: *Total number of disk requests and disk bytes moved in the query workload on an Intel 520 SSD for five tested file systems.*

on the limited structure in file systems, hiding structures inside directories meant to represent files [6, 14, 15, 21, 52], even though this may increase the number of small files in the file system. In this paper, we return to the basic premise, file system metadata is a natural fit for table-based representation, and show that today's lightweight data stores may be up to the task. We are concerned with an efficient representation of huge numbers of small files, not strengthening transactional semantics [16, 19, 25, 41, 46, 51].

Early file systems stored directory entries in a linear array in a file and inodes in simple on-disk tables, separate from the data of each file. Clustering within a file was pursued aggressively, but for different files clustering was at the granularity of the same cylinder group. It has long been recognized that small files can be packed into the block pointer space in inodes [30]. C-FFS [12] takes packing further and clusters small files, inodes and their parent directory's entries in the same disk readahead unit, the track. A variation on clustering for efficient prefetching is replication of inode fields in directory entries, as is done in NTFS[9]. TABLEFS pursues an aggressive clustering strategy; each row of a table is ordered in the table with its parent directory, embedding directory entries, inode attributes and the data of small files. This clustering manifests as adjacency for objects in the lower level object store if these entries were created/updated close together in time, or after compaction has merge sorted them back together.

Beginning with the Log-Structured File System (LFS)[38], file systems have exploited write allocation methods that are non-overwrite, log-based and deferred. Variations of log structuring have been implemented in NetApp's WAFL, Sun's ZFS and BSD UNIX [3, 18, 45]. In this paper we are primarily concerned with the disk access performance implications of non-overwrite and log-based writing, although the potential of strictly ordered logging to simplify failure recovery in LFS has been emphasized and compared to various write ordering schemes such as Soft Updates and Xsyncfs [28, 31, 44]. LevelDB's recovery provisions are consistent with delayed periodic journalling, but could be made consistent with stronger ordering schemes. It is worth noting that the design goals of Btrfs call for non-overwrite (copy-on-write) updates

to be clustered and written sequentially[37], largely the same goals of LevelDB in TABLEFS. Our measurements indicate, however, that the Btrfs implementation ends up doing far more small disk accesses in metadata dominant workloads.

Partitioning the contents of a file system into two groups, a set of large file objects and all of the metadata and small files, has been explored in hFS[53]. In their design large file objects do not float as they are modified, while the metadata and small files are log structured. TABLEFS has this split as well, with large file objects handled directly by the backing object store, and metadata updates approximately log structured in LevelDB's partitioned LSM tree. However, TABLEFS does not handle disk allocation, relying entirely on the backing object store to handle large objects well.

Log-Structured Merge trees [33] were inspired in part by LFS, but focus on representing a large B-tree of small entries in a set of on-disk B-trees constructed of recent changes and merging these on-disk B-trees as needed for lookup reads or in order to merge on-disk trees to reduce the number of future lookup reads. LevelDB [23] and TokuFS [11] are LSM trees. Both are partitioned in that the on-disk B-trees produced by compaction cover small fractions of the key space, to reduce unnecessary lookup reads. TokuFS names its implementation a Fractal Tree, also called streaming B-trees[5], and its compaction may lead to more efficient range queries than LevelDB's LSM tree because LevelDB uses Bloom filters[7] to limit lookup reads, a technique appropriate for point lookups only. If bounding the variance on insert response time is critical, compaction algorithms can be more carefully scheduled, as is done in bLSM[43]. TABLEFS may benefit from all of these improvements to LevelDB's compaction algorithms, which we observe to sometimes consume disk bandwidth injudiciously (See Section 4.3).

Recently, VT-trees [46] were developed as a modification to LSM trees to avoid always copying old SSTable content into new SSTables during compaction. These trees add another layer of pointers so new SSTables can point to regions of old SSTables, reducing data copying but requiring extra seeks and eventual defragmentation.

6 Conclusion

File systems for magnetic disks have long suffered low performance when accessing huge collections of small files because of slow random disk seeks. TABLEFS uses modern key-value store techniques to pack small things (directory entries, inode attributes, small file data) into large on-disk files with the goal of suffering fewer seeks when seeks are unavoidable. Our implementation, even hampered by FUSE overhead, LevelDB code overhead, LevelDB compaction overhead, and pessimistically padded inode attributes, performs as much as 10 times better than state-of-the-art local file systems in extensive metadata update workloads.

References

[1] FUSE. http://fuse.sourceforge.net/.

[2] Memcached. http://memcached.org/.

[3] ZFS. http://www.opensolaris.org/os/community/zfs.

[4] Nitin Agrawal, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Generating realistic impressions for file-system benchmarking. In *Proccedings of the 7th conference on File and storage technologies (FAST)*, 2009.

[5] Michael A. Bender, Martin Farach-Colton, Jeremy T. Fineman, Yonatan R. Fogel, Bradley C. Kuszmaul, and Jelani Nelson. Cache-oblivious streaming B-trees. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures (SPAA)*, 2007.

[6] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. PLFS: a checkpoint filesystem for parallel applications. In *Proceedings of the ACM/IEEE conference on Supercomputing*, 2009.

[7] B.H. BLOOM. Space/time trade-offs in hash coding with allowable errors. *Communication of ACM 13, 7*, 1970.

[8] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, 2006.

[9] H. Custer. Inside the windows NT file system. *Microsoft Press*, 1994.

[10] Shobhit Dayal. Characterizing HEC storage systems at rest. In *Carnegie Mellon University, Technical Report CMU-PDL-08-109*, 2008.

[11] John Esmet, Michael Bender, Martin Farach-Colton, and Bradley Kuszmaul. The TokuFS streaming file system. *Proceedings of the USENIX conference on Hot Topics in Storage and File Systems*, 2012.

[12] Gregory R. Ganger and M. Frans Kaashoek. Embedded inodes and explicit grouping: Exploiting disk bandwidth for small files. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, 1997.

[13] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of the 19th ACM symposium on Operating systems principles*, 2003.

[14] Michael Austin Halcrow. eCryptfs: An Enterprise-class Encrypted Filesystem for Linux. *Proc. of the Linux Symposium, Ottawa, Canada*, 2005.

[15] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. A file is not a file: understanding the I/O behavior of Apple desktop applications. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011.

[16] R. Haskin, Y. Malachi, W. Sawdon, and G. Chan. Recovery management in quicksilver. *In Proceedings of the Eleventh ACM Symposium on Operating System Principles*, 1987.

[17] HDFS. Hadoop file system. http://hadoop.apache.org/.

[18] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. In *USENIX Winter Technical Conference*, 1994.

[19] Aditya Kashyap. File system extensibility and reliability using an in-kernel database. *Master Thesis, Computer Science Department, Stony Brook University*, 2004.

[20] Jeffrey Katcher. Postmark: A new file system benchmark. In *NetApp Technical Report TR3022*, 1997.

[21] Hyojun Kim, Nitin Agrawal, and Cristian Ungureanu. Revisiting storage for smartphones. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, 2012.

[22] Jan Kra. Ext4, BTRFS, and the others. In *Proceeding of Linux-Kongress and OpenSolaris Developer Conference*, 2009.

[23] LevelDB. A fast and lightweight key/value database library. http://code.google.com/p/leveldb/.

[24] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. SILT: a memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011.

[25] Barbara Liskov and Rodrigo Rodrigues. Transactional file systems can be fast. *Proceedings of the 11th ACM SIGOPS European Workshop*, 2004.

[26] Lustre. Lustre file system. http://www.lustre.org/.

[27] Avantika Mathur, Mingming Cao, and Suparna Bhattacharya. The new Ext4 lesystem: current status and future plans. In *Ottawa Linux Symposium*, 2007.

[28] Marshall Kirk McKusick and Gregory R. Ganger. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. *Proceedings of the annual conference on USENIX Annual Technical Conference, FREENIX Track*, 1999.

[29] Dutch T. Meyer and William J. Bolosky. A study of practical deduplication. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, 2011.

[30] Sape J. Mullender and Andrew S. Tanenbaum. Immediate files. *SoftwarePractice and Experience*, 1984.

[31] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. Rethink the sync. *ACM Transactions on Computer Systems, Vol.26, No.3 Article 6*, 2008.

[32] Michael A. Olson. The design and implementation of the Inversion file system. In *USENIX Winter Technical Conference*, 1993.

[33] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 1996.

[34] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast crash recovery in RAMCloud. In *Proceedings of the 23rd ACM symposium on Operating systems principles*, 2011.

[35] Swapnil Patil and Garth Gibson. Scale and concurrency of GIGA+: File system directories with millions of files. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, 2011.

[36] Ohad Rodeh. B-trees, shadowing, and clones. *Transactions on Storage*, 2008.

[37] Ohad Rodeh, Josef Bacik, and Chris Mason. BRTFS: The Linux B-tree Filesystem. *IBM Research Report RJ10501 (ALM1207-004)*, 2012.

[38] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the thirteenth ACM symposium on Operating systems principles*, 1991.

[39] Robert Ross and Robert Latham. PVFS: a parallel file system. In *Proceedings of the ACM/IEEE conference on Supercomputing*, 2006.

[40] Frank B. Schmuck and Roger L. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX conference on File and storage technologies*, 2002.

[41] Russell Sears and Eric A. Brewer. Stasis: Flexible transactional storage. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, 2006.

[42] Russell Sears, Catharine Van Ingen, and Jim Gray. To BLOB or Not To BLOB: Large Object Storage in a Database or a Filesystem? *Microsoft Technical Report*, 2007.

[43] Russell Sears and Raghu Ramakrishnan. bLSM: a general purpose log structured merge tree. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2012.

[44] Margo Seltzer, Gregory Ganger, Kirk McKusick, Keith Smith, Craig Soules, and Christopher Stein. Journaling versus soft updates: Asynchronous meta-data protection in file systems. *Proceedings of the annual conference on USENIX Annual Technical Conference*, 2000.

[45] Margo I. Seltzer, Keith Bostic, Marshall Kirk McKusick, and Carl Staelin. An implementation of a log-structured file system for UNIX. *USENIX Winter Technical Conference*, 1993.

[46] Pradeep Shetty, Richard Spillane, Ravikant Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. Building workload-independent storage with VT-Trees. In *Proccedings of the 11th conference on File and storage technologies (FAST)*, 2013.

[47] Michael Stonebraker. Operating System Support for Database Management. *Commun. ACM*, 1981.

[48] Adam Sweeney. Scalability in the XFS file system. In *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, 1996.

[49] Brent Welch, Marc Unangst, Zainul Abbasi, Garth Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable performance of the panasas parallel file system. In *Proceedings of the 6th USENIX conference on File and Storage Technologies*, 2008.

[50] Ric Wheeler. One billion files: pushing scalability limits of linux filesystem. In *Linux Foudation Events*, 2010.

[51] CHARLES P. WRIGHT, RICHARD SPILLANE, GOPALAN SIVATHANU, and EREZ ZADOK. Extending ACID Semantics to the File System. *ACM Transactions on Storage*, 2007.

[52] Erez Zadok and Jason Nieh. FiST: A Language for Stackable File Systems. *Proceedings of the annual conference on USENIX Annual Technical Conference*, 2000.

[53] Zhihui Zhang and Kanad Ghose. hFS: A hybrid file system prototype for improving small file and metadata performance. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2007.