

TABLEFS: Embedding a NoSQL Database Inside the Local File System

Kai Ren and Garth Gibson
Carnegie Mellon University {kair, garth}@cs.cmu.edu

Conventional file systems are optimized for large file transfers instead of workloads that are dominated by metadata and small file accesses. This paper examines using techniques adopted from NoSQL databases to manage file system metadata and small files, which feature high rates of change and efficient out-of-core data representation. A FUSE file system prototype was built by storing file system metadata and small files into a modern key-value store: LevelDB. We demonstrate that such techniques can improve the performance of modern local file systems in Linux for workloads dominated by metadata and tiny files.

Index Term – System software, File systems, Metadata representation, Log-structure Merge Tree.

INTRODUCTION

While parallel and Internet service file systems have demonstrated effective scaling for high bandwidth, large file transfers in the last decade [7], [9], [15], [24], [25], [32], the same is not true for workloads that are dominated by metadata and tiny file access [21], [33]. Instead there has emerged a large class of scalable small-data storage systems, commonly called key-value stores, that emphasize simple (NoSQL) interfaces and large in-memory caches [14], [19].

Some of these key-value stores feature high rates of change and efficient out-of-memory log-structured merge (LSM) tree structures [28], [20], [4]. We assert that file systems should adopt techniques from modern key-value stores for metadata and tiny files, because these systems are “thin” enough to provide the performance levels required by file systems [30]. We are not attempting to improve semantics (e.g. transactions [11], [26]).

To motivate our assertion, in this paper we present experiments in the most mature and restrictive of environments: a local file system managing one magnetic hard disk. 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 by adding an interposed file system layer [1] that represents metadata and tiny files in a LevelDB key-value store [13] that stores its LSM tree and write-ahead log segments in these same local file systems.

I. BACKGROUND

Even in the era of big data, most things in a file system are small [5], [17]. Inevitably, scalable systems should

expect the numbers of small files to soon achieve and exceed billions, a known problem for both the largest [21] and most local file systems [33].

A. Embedded Databases

File system metadata is structured data, a natural fit for relational database techniques. However, because of large size, complexity and slow speed, file system developers have long been reluctant to incorporate traditional databases into the lower levels of file systems [18], [29]. Modern stacked file systems often expand on the limited structure in file systems, hiding structures inside directories meant to represent files [3], [8], [12], although 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 natural 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 [11], [26].

B. Local File System Techniques

Early file systems stored directory entries in a linear array in a file and inodes in simple on-disk tables. Modern file systems such as Ext4 uses hash tables, and XFS, ZFS, and Btrfs use B-Trees, for indexing directories [16], [22], [31]. Moreover, LFS, WAFL, ZFS and Btrfs [2], [10], [23] use non-overwrite or log structured methods to batch metadata changes and write them to disk sequentially. Such techniques may in many cases group all the metadata needed to access a file together on-disk by exploiting temporal locality. C-FFS [6], however, explicitly groups the inodes of files with

their directory entries, and small files from the same directory in adjacent data blocks. And hFS [34] uses log structuring to manage metadata and update-in-place to manage large files.

C. LevelDB and LSM Trees

LevelDB [13] is an open-source key-value storage library that features Log-Structured Merge (LSM) Trees [20], which were popularized by BigTable [4]. It provides simple APIs such as GET, PUT, DELETE and SCAN. 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.

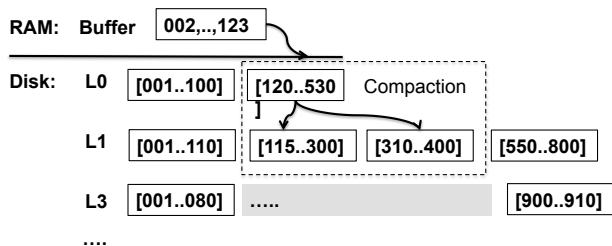


Fig. 1. LevelDB (a variant of the Log-structure Merge Tree) consists of multiple SSTables that store sorted key-value pairs. SSTables are grouped into different levels. Lower levels contain more recently inserted key-value pairs.

The basic technique used by LSM Trees and LevelDB is to manage multiple large sorted arrays of on-disk data (called SSTables) in a log-structured way. Figure 1 illustrates the basic in-memory/disk data layout of SSTables. These SSTables are grouped into several levels numbered starting from 0. When inserting or updating, elements are initially write-back buffered in memory. When the memory buffer exceeds a threshold (4MB by default), the buffer is dumped into disk as a SSTable. Level 0 of LevelDB contains the most recently dumped SSTables. The higher levels contain older data. When querying an element, it requires searching for the element level by level, and starts from Level 0. It returns the first matched key-value pair, which is the most up-to-date version. Another invariant maintained by LevelDB is that SSTables in the same level have disjoint key ranges. So querying an element only needs to read at most one SSTable at each level above level 0. To further reduce the number of SSTables it searches, LevelDB also maintains a memory index that records the key range of each SSTable and uses bloom-filters to reduce unnecessary lookups. To improve read query speed and remove deleted data, it periodically merge-sorts a list of SSTables. This process is called “compaction”, and is similar to *online defragmentation* in file systems, and

cleaning in log-structured file system [23]. During compaction, LevelDB picks an SSTable from some level L , and all SSTables from level $L + 1$ that have overlapping key ranges. It reads the selected files, and replace disjoint range of SSTables generated from the merge sort of the selected set of SSTables with another set.

II. 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 and represents directories, inodes and small files in one all encompassing table. TABLEFS only writes to the local disk large objects such as write-ahead logs, SSTables containing changes to the metadata table, and files whose size is large.

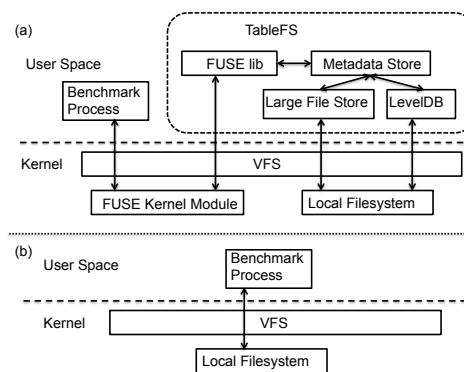


Fig. 2. (a) The architecture of TABLEFS. A FUSE kernel module redirects file system calls to TABLEFS, and TABLEFS stores objects into either LevelDB or a large file store. (b) The case architecture our experiments compare against in Section III. These figures suggest the large and “unfair” overhead TABLEFS experiences relative to the traditional local file systems.

A. 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 of about 2MB each, the local file system sees many fewer, larger objects.

B. Large File “Blob” Store

Files larger than T bytes are stored directly in the object store 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 the limited scalability of directory entries in some file systems. In TABLEFS today, T , the threshold for blobbing a file is 4KB, which is the median size of files in desktop workloads [17], although others have suggested T be 256KB to 1MB [27].

C. Table Schema

TABLEFS’s metadata store aggregates directory entries, inode attributes and small files into one LevelDB table with a row for each file. Each file is given an inode number 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, 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.

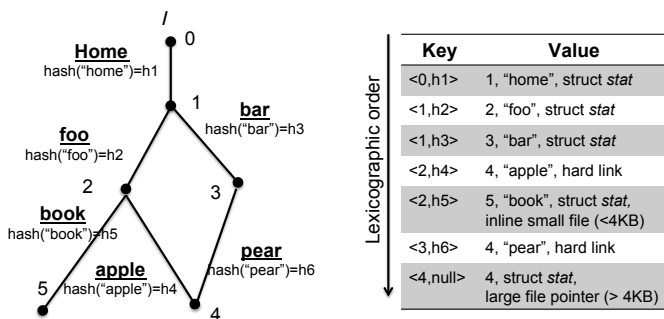


Fig. 3. An example illustrating the table schema used by TABLEFS’s metadata store. The file with inode number 4 has two hard links, one called “apple” from directory *foo* and the other called “pear” from directory *bar*.

All the entries in the same directory have rows that share the same first 64 bits in their the table’s 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’s 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.

D. Hard Link

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

a hard link. If one of hard links is deleted, only the row containing the corresponding hard link is deleted, and the row with its own inode number and other rows with hard links are still kept in table without modification.

E. 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 with 32 bit inode numbers will require frequent inode number recycling, a problem beyond the scope of this paper and shared by many file systems.

F. Locking and Consistency

LevelDB provides atomic batch insert 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 database 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 inode attributes are stored in one 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 accesses.

G. 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 synchronously every five seconds.

III. EVALUATION

A. Evaluation System

We evaluate our TABLEFS prototype with a Linux desktop computer equipped as follows:

Linux Ubuntu 10.04, Kernel 2.6.32-33
 CPU Intel Core2 Quad Q9550 @ 2.83GHz
 DRAM DDR SDRAM 4GB, using only 512 MB
 Hard Disk Seagate ST31000340NS
 SATA, 7200rpm, 1TB
 Using only a 5GB partition
 Random Seeks 145 seeks/sec peak
 Sequential Reads 121.6 MB/sec peak
 Sequential Writes 106.4 MB/sec peak

We limit the machine’s available memory to only 512 MB (setting boot parameters of Linux), to prohibit any cache in a user process or kernel cache from using much more memory than is available to another file system, because we cannot easily control all cache sizes for internal file systems.

We compare TABLEFS with Linux’s most sophisticated local file systems: Ext4, XFS, and BTRFS, whose versions are 1.41.11, 3.1.0, and 0.19 respectively. Ext4 is mounted with “ordered” journaling to force all data to be flushed out to disk before its metadata is committed to the journal. We believe this is the same fault semantics we achieve in TABLEFS. By default, the journal of Ext4 is synchronously committed to disks every five seconds. XFS and BTRFS uses similar policies to synchronously update journals.

BTRFS, by default, duplicates metadata and also calculates checksums for data and metadata. We disable both features (unavailable in the other file systems) when benchmarking BTRFS. TABLEFS always uses BTRFS as the underlying file system. Since the tested filesystems have different inode sizes (Ext4 and XFS use 256 bytes and BTRFS uses 136 bytes), we pessimistically punish TABLEFS by padding its inode attributes to 256 bytes. This slows down TABLEFS quite a bit, but it still performs quite well.

All benchmarks are simple “create and query” micro-benchmarks intended only to show that even with the overhead of FUSE, LevelDB, LevelDB compaction, and padded inode structures, TABLEFS can improve performance on the local file system.

B. Benchmark with Metadata Only

We first micro-benchmark the efficiency of pure metadata operations. The micro-benchmark consists of two phases. The first phase (“creation”) generates a file system of one million files, all zero length. This file system has the same namespace as one author’s personal Ubuntu desktop, trimmed back to one million files. The benchmark creates this test namespace in the tested file systems in depth first order. The second phase (“query”)

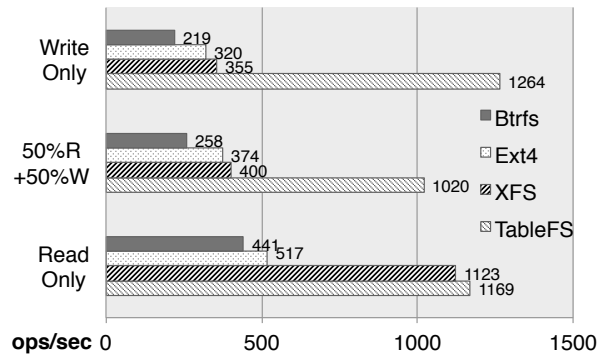


Fig. 4. Performance of each file system in the query phase of the metadata-only benchmark.

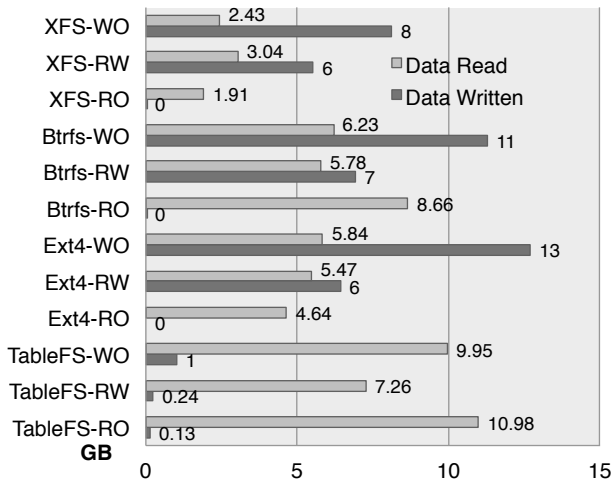
issues 2 million random read or write queries to random (uniform) files or directories. A read query calls *stat* on the file, a write query randomly does either a *chmod* or *utime* to update the mode or the timestamp fields. Between the two phases, we force local filesystems to drop their cache, so that the second phases starts with a cold cache.

Figure 4 shows the performance in operations per second, averaged over the query phase, for three different ratios of read and write queries: (1) read-only queries, (2) 50% read and 50% write queries, and (3) write-only queries. TABLEFS is almost 2.5X to 3X faster than the other tested file systems in workloads having writes and it achieves comparable performance in read-only workload.

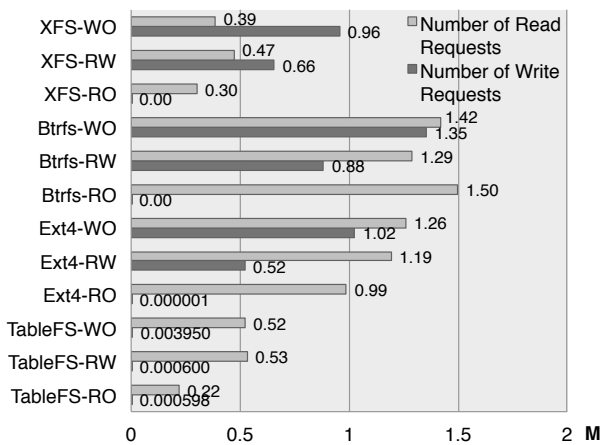
Figure 5 shows the total disk traffic (total size and requests) during the query phase in the three workloads. These numbers are extracted from Linux *proc* file system (*/proc/diskstats*). Compared to other file systems, TABLEFS reduces write disk traffic (the number of write requests) significantly. This shows that using LevelDB effectively batches small random writes into large sequential writes. For read requests, with bloom filtering and an effective memory index, TABLEFS achieves a similar low number of read requests, although because of compaction the total amount of data read is large.

Figure 5 also shows that TABLEFS incurs write traffic in read-only workloads. This is due to compaction in the underlying LevelDB SSTables. LevelDB maintains an individual counter of false-positive lookups in each SSTable. If one SSTable receives too many false-positive lookups, a compaction will be triggered to merge this SSTable with other SSTables within the same key range to reduce false-positive lookups.

Figure 6 shows a behavior timeline for TABLEFS during the query phase of the 50%Read-50%Write workload. The throughput spike in the beginning of that test is



(a) Total data read from / written to disk (in gigabytes)



(b) Total number of disk read / write requests (in millions)

Fig. 5. Total disk traffic during the query phase of metadata-only benchmark for three workloads

due to everything fitting in the cache. Later in the query phase, there are two drops in the system throughput, corresponding to spikes in the disk read and write traffic. This behavior is caused by compactions in LevelDB, in which SSTables are merged and sequentially written back to disks.

C. Benchmark with Small Files

The second micro-benchmark is similar to first except that we create one million 1KB files in 1000 directories, each directory containing 1000 files. In the query phase, read queries retrieve the content of a file and write queries overwrite the whole file. Files in the query phase are still randomly picked, and distributed uniformly in the namespace. Figure 7 shows the results with a 50%Read-50%Write workload of one million queries. In creation phase, TABLEFS is much slower than Ext4 and BTRFS. This is because the FUSE overhead is more

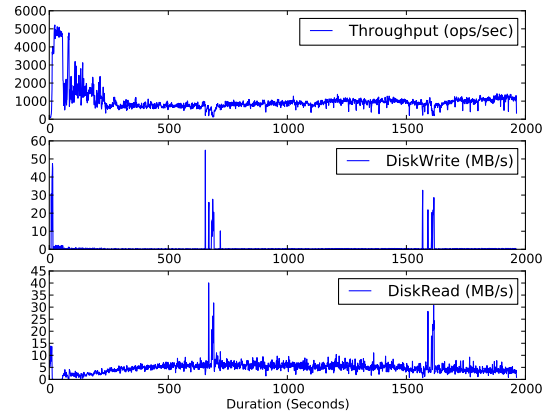


Fig. 6. TABLEFS’s system throughput and the underlying disk traffic during the query phase of metadata-only benchmark, when 50% of queries are reads and 50% are writes. The average disk write throughput is low about 3 MB/s. The data is sampled in every second.

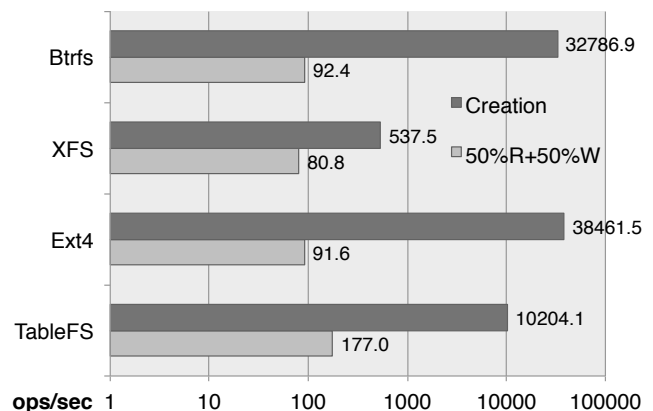


Fig. 7. Performance during the query phase of small-files benchmark. The workload is 1M 50%Read-50%Write queries on 1M 1KB files.

significant with non-zero file sizes. In the query phase, however, TABLEFS outperforms all other file systems by 2X. We cannot estimate how much faster TABLEFS would be without FUSE overhead, but prior experiments suggest it can be large [3].

IV. CONCLUSION

File systems have long suffered low performance when accessing huge collections of small files because caches cannot hide all 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 much better than state-of-the-art

local file systems when the workload is pure metadata and much better during the query phase for small file workloads.

REFERENCES

- [1] FUSE. <http://fuse.sourceforge.net/>.
- [2] ZFS. <http://www.opensolaris.org/os/community/zfs>.
- [3] John Bent and et al. PLFS: a checkpoint filesystem for parallel applications. In *SC*, 2009.
- [4] Fay Chang and et al. Bigtable: a distributed storage system for structured data. In *OSDI*, 2006.
- [5] Shobhit Dayal. Characterizing HEC storage systems at rest. Technical report, Carnegie Mellon University, 2008.
- [6] Gregory R. Ganger and M. Frans Kaashoek. Embedded inodes and explicit grouping: Exploiting disk bandwidth for small files. In *USENIX ATC*, 1997.
- [7] Sanjay Ghemawat and et al. The Google file system. In *SOSP*, 2003.
- [8] Tyler Harter and et al. A file is not a file: understanding the I/O behavior of Apple desktop applications. In *SOSP*, 2011.
- [9] HDFS. Hadoop file system. <http://hadoop.apache.org/>.
- [10] Dave Hitz and et al. File system design for an NFS file server appliance. In *USENIX Winter*, 1994.
- [11] Aditya Kashyap and et al. File system extensibility and reliability using an in-kernel database. *Master Thesis, Computer Science Department, Stony Brook University*, 2004.
- [12] Hyojun Kim and et al. Revisiting storage for smartphones. In *FAST*, 2012.
- [13] LevelDB. A fast and lightweight key/value database library. <http://code.google.com/p/leveldb/>.
- [14] Hyeontaek Lim and et al. SILT: a memory-efficient, high-performance key-value store. In *SOSP*, 2011.
- [15] Lustre. Lustre file system. <http://www.lustre.org/>.
- [16] Avantika Mathur and et al. The new ext4 filesystem: current status and future plans. In *Ottawa Linux Symposium*, 2007.
- [17] Dutch T. Meyer and William J. Bolosky. A study of practical deduplication. In *FAST*, 2011.
- [18] Michael A. Olson. The design and implementation of the inversion file system. In *USENIX Winter*, 1993.
- [19] Diego Ongaro and et al. Fast crash recovery in ramcloud. In *SOSP*, 2011.
- [20] Patrick O’Neil and et al. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 1996.
- [21] Swapnil Patil and Garth A. Gibson. Scale and concurrency of GIGA+: File system directories with millions of files. In *FAST*, 2011.
- [22] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *IBM Research Report RJ10501 (ALM1207-004)*, 2012.
- [23] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *SOSP*, 1991.
- [24] Robert B. Ross and et al. PVFS: a parallel file system. In *SC*, 2006.
- [25] Frank B. Schmuck and Roger L. Haskin. GPFS: A shared-disk file system for large computing clusters. In *FAST*, 2002.
- [26] Russell Sears and Eric A. Brewer. Stasis: Flexible transactional storage. In *OSDI*, 2006.
- [27] Russell Sears and et al. To blob or not to blob: Large object storage in a database or a filesystem? *Microsoft Technique Report*, 2007.
- [28] Jan Stender and et al. BabuDB: Fast and efficient file system metadata storage. In *SNAPI ’10*, 2010.
- [29] Michael Stonebraker. Operating system support for database management. *Commun. ACM*, 1981.
- [30] Michael Stonebraker and Ugur Çetintemel. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *ICDE*, 2005.
- [31] Adam Sweeney. Scalability in the xfs file system. In *USENIX ATC*, 1996.
- [32] Brent Welch and et al. Scalable performance of the panasas parallel file system. In *FAST*, 2008.
- [33] Ric Wheeler. One billions files: pushing scalability limits of linux filesystem. In *Linux Foundation Events*, 2010.
- [34] Zihui Zhang and et al. hFS: A hybrid file system prototype for improving small file and metadata performance. In *EuroSys*, 2007.