LinLogFS — A Log-Structured Filesystem For Linux

Christian Czezatke
xS+S

M. Anton Ertl
TU Wien

{czezatke,anton}@mips.complang.tuwien.ac.at
http://www.complang.tuwien.ac.at/projects/linux.html

Abstract

LinLogFS is a log-structured filesystem for Linux. It currently offers features like fast crash recovery and in-order write semantics. We implemented LinLogFS by putting a logging layer between an adapted version of the ext2 file system and the block device.

1 Introduction

In early 1998 we started working on LinLogFS[1] to address the two main shortcomings that Linux had (in comparison to Digital/Tru64 Unix with AdvFS):

- Crash recovery is slow and sometimes requires manual intervention.
- There is no way to make *consistent* backups while the file system is writable.

These shortcomings are not just inconvenient for the users of desktop machines, they are also one of the main barriers against Linux servers in applications requiring high availability.

We decided to solve these problems by implementing a log-structured file system; this approach also promises a number of other benefits:

- In-order semantics for data consistency upon system failure (see Section 3.2).
- Cheap cloning (writable snapshots) of the whole file system can be used for exploring what-if scenarios while writing to the other clone of the file system (in addition to consistent backups).
- Growing, shrinking, or migrating a file system while it is mounted.
- Fast update of a network mirror after a network failure.
- Relatively good performance for synchronous writes (NFS).
- Good write performance with RAIDs.

In the meantime, there are a number of Linux projects nearing completion that address fast crash recovery and consistent backups (see Section 8), and may also provide some of the other advantages. However, we think that LinLogFS provides an interesting design point, and that its features combined with its performance will make it attractive once it is mature.

We first introduce log-structured file systems (Section 2), then discuss some advanced topics in the design of log-structured file systems (Section 3). In Section 4 we describe which of these ideas we have implemented until now. Then we describe our implementation, first the layering (Section 5), then the modifications to the Linux kernel (Section 6). In Section 7 we compare the performance of LinLogFS and other Linux file systems. In Section 8 we discuss related work.

2 A Log-Structured File System

This section explains the concept of log-structured file systems by using LinLogFS as an example. Most of this description also fits other log-structured file systems like BSD LFS. Section 8 describes the differences from other systems and the original points in LinLogFS.

A file system provides a layer of abstraction that allows the user to deal with files and directories on top of a block-oriented storage medium. In Unix the

[1] Formerly known as dtfs (renamed to avoid conflicts with SCO's dtfs).

Paper and BibTeX entry are available at http://www.complang.tuwien.ac.at/papers/. This paper was published in:
Usenix 2000 Freenix Track, pages 77-88

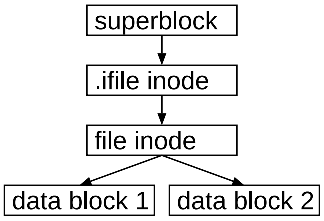

Figure 1: On-disk structure for locating data blocks of a file

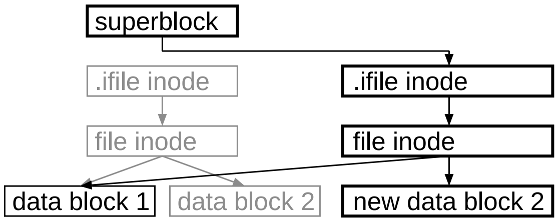

Figure 2: Change of the on-disk structure when overwriting data block 2 of the file

file system can be considered as consisting of two layers: the upper layer maps the hierarchical structure of the directory tree onto the flat namespace of inode numbers; the lower layer maps an inode number and a position within a file to the disk block that contains the data. The difference between log-structured and other file systems is in the lower layer, so we will focus on it in this section (and in the rest of this paper).

2.1 Reading

Reading is quite similar to conventional file systems like ext2 and BSD FFS: when reading a certain block from a certain file, the file's inode is accessed, and the data block is accessed by following the pointer there (possibly going through several levels of indirect blocks).

How do we find the inode? In LinLogFS (in contrast to conventional file systems) the inodes reside in the file `.ifile`. We find the inode of this file through the superblock (see Fig. 1).

2.2 Writing

The distinctive feature of log-structured file systems is that they perform no update-in-place; instead they write changed data and meta-data to an unused place on the disk. So, when writing a block of data to a file, the block is written to a new location, the now-changed indirect block pointing to it is written to a new location, and so on for the whole chain from the superblock to the data block (see Fig. 2).[2]

Benefit: Since existing data is not destroyed by writing over it, it is easy to clone the file system for backup purposes and to undelete files.

The superblock is written and updated in a fixed place, allowing us to find it, the `.ifile` and thus the whole file system when mounting it. Therefore, updating the superblock commits all the writes since the last superblock update.

Benefit: Having explicit commits allows performing atomic operations involving more than one block, e.g., directory operations. This makes it easy to ensure that the committed state of the file system is consistent and provides fast crash recovery. It also makes it easy to guarantee in-order write semantics upon a crash (see Section 3.2).

Since there is no restriction on where the blocks must be written, they can all be appended to a log. Most of the time a number of writes can be collected and written in one large batch. Ideally the disk is written from the start to the end (see Section 2.3 for deviations from this ideal).

Benefit: The resulting sequential writes require no disk seeks (except for the superblock updates; see Section 3.1.1 for reducing these seeks) and thus allow fast synchronous writes. Large sequential writes are also helpful when writing to RAIDs. Moreover, they make it easy to find the newly written data when synchronizing a (network) mirror.

2.3 Reclaiming Space

Unfortunately disks are finite, so we have to reclaim the space occupied by deleted or overwritten data before the disk is full. LinLogFS divides the volume into segments of about 512KB.[3] The *cleaner* program reclaims a segment by copying the (hopefully small amount of) live data to the end of the log.

[2] This is similar to the handling of data structures in single-assignment languages, in particular eager functional programming languages like ML.

[3] Note that starting a new segment does not commit changes written in the last one (at least in LinLogFS).

Then the reclaimed segment is available again for writing.

Benefit: Using large segments instead of a free blocks map ensures that the filesystem can usually write sequentially, avoiding seeks. Using segments instead of treating the log as one big ring buffer allows segregating long-lived from short-lived data and varying the cleaning frequency accordingly. The copying approach of the cleaner makes it easy to free specific segments in order to shrink or migrate a volume.

The cleaner uses segment summary information: For every block in the segment, the segment summary contains the file and the block number within the file. This makes it efficient for the cleaner to find out whether a block is live.

3 Log-structured File System Details

This section discusses a number of design issues for log-structured file systems in more detail.

3.1 Optimizations

3.1.1 Roll forward

Writing the superblock requires a seek to the superblock and a seek back to the log, limiting the performance for small synchronous writes.

This can be solved by writing special commit blocks into the log. When mounting a file system, the file systems does not assume that the superblock points to the end of the log. Instead, it uses the pointed-to block as a roll-forward start point and scans forward for the last written commit block. The file system uses that commit block as the end of the log; i.e., it contains the .ifile inode and allows finding the whole file system. To limit the crash recovery time, the superblock is still written now and then (e.g., when about 10MB of segments have been written to achieve a crash recovery time on the order of one second).

3.1.2 Change Records

Instead of writing every changed block out when committing, space and write time can be saved in many cases by recording just what has changed. During roll-forward, the file system updates its in-memory copies of the changed (meta-)data with this information. Of course, the changed blocks still have to be written out when establishing a new roll-forward start point, but for blocks that are changed several times between such start points, this optimization pays off.

The state to which a change record is applied on recovery is known completely, so we can use any kind of change record (in contrast to journaling file systems, where change records are usually restricted to idempotent operations [VGT95]).

Change records influence recovery time by requiring reading the blocks that they change (and possibly indirect blocks to find them). This should be taken into account when deciding when to write a new roll-forward start point. Another consequence is that the cleaner has to treat these blocks as live.

3.1.3 Block Pointers

Writing one data block can cause up to eight meta-data blocks to be written to the log: up to three indirect blocks, the block containing the inode of the file, up to three indirect blocks of the `.ifile`, and the commit block containing the `.ifile` inode; the reason for this cascade is that the pointer to the next block in the chain changes. The usual case is much better, because usually many written data blocks share meta-data blocks. However, for small synchronous writes block pointer updates would be a problem. Moreover, part of this meta-data (the .ifile inode and its indirect blocks) tends to be overwritten and thus become dead quickly, requiring a cleaning pass even if none of the user data in the segment has died.

To alleviate this problem, we are considering to add change records for the most frequent cause of meta-data updates: updates of pointers to blocks. The change record would describe which block of which file now resides where.

There is an improvement of this optimization: we need to store the information about which blocks reside where only once, not once for the change records and once as segment summary information for the cleaner.

Another refinement of this optimization is to combine the change records for ranges of blocks of the same file into one change record, saving space.

3.2 In-order semantics

With *in-order semantics* we mean that the state of the file system after recovery represents all write()s (or other changes) that occurred before a specific point in time, and no write() (or other change) that occurred afterwards. I.e., at most you lose a minute or so of work.

The value of this guarantee may not be immediately obvious. It means that if an application ensures file data consistency in case of its own unexpected termination by performing writes and other file changes in the right order, its file data will also be consistent (but possibly not up-to-date) in case of a system failure; this is sufficient for many applications.

However, most file systems nowadays guarantee only meta-data consistency and require the extensive use of fsync(2) to ensure any data consistency at all. So, if applications fsync, isn't the in-order guarantee worthless? Our experience (http://www.complang.tuwien.ac.at/anton/sync-metadata-updates.html) suggests that even popular applications like Emacs don't fsync enough to avoid significant data losses; and even for applications that try to ensure data consistency across OS crashes with fsync, this is probably not a very well-tested feature. So, providing the in-order guarantee will improve the behaviour of many applications.

Moreover, fsync() is an expensive feature that should not be used when cheaper features like the in-order guarantee are sufficient.

One problem with guaranteeing in-order semantics in a log-structured file system is that frequent fsyncs can lead to fragmenting unrelated files.[4] There are two possible solutions:

- Defragment the files upon cleaning.

- Weaken the guarantee such that the state of the file system after recovery represents all write()s (or other changes) that occurred before a specific point in time, and only fsync()s that occurred afterwards; after all, applications using fsync supposedly know what they are doing.

[4]The reason for this fragmentation is: with in-order semantics, fsync() is equivalent to sync(), i.e., all changes are written out. Now assume a workload where lots of files are written concurrently. Given the sequential writing strategy of log-structured file systems, parts of unrelated files end up close to each other, and these files will be fragmented.

3.3 Cleaning and Cloning

We have not implemented a cleaner for LinLogFS yet; this section reflects our current thoughts on cleaning.

The cleaning heuristics discussed in the literature [RO92, BHS95] work well, so this is the direction we would like to take: cleaning proactively when the disk is idle, and using utilization and age for choosing segments to clean. We also like the Spiralog [WBW96] idea of producing entire segments out of collected data (instead of copying the collected data into the current segment), which allows avoiding mixing up data with different ages (i.e., different expected lifetimes). The cleaner can also be used for defragmentation.

The main new problem in cleaning LinLogFS is how to deal with cloning. The cleaners described in the literature use information about the utilization of a segment in their heuristics. This information is not easily available in the presence of cloning; in particular, dismissing a clone can result in changes in utilization that are expensive to track. File deletion would also be relatively expensive (especially considering that it would be very cheap otherwise): for every block in the file the filesystem would have to check whether the block is still alive in any other clone.

Therefore, we have to search for heuristics that work without this information, or with just approximate information (like upper and lower bounds on utilization).

If we don't find such heuristics, we may have to bite the bullet, and keep track of which blocks are alive in which clone. WAFL [HLM94] uses a simple bit matrix for this (with 32 bits/block). This has the following disadvantages:

RAM consumption For a 20GB file system with 4KB blocks, this requires 20MB, much of which will have to reside in RAM at most times, in particular when cloning or dismissing clones. This is especially worrying because disks tend to grow faster than RAM since the early 1990s.

High cloning and dismissing cost Cloning requires copying a bit to another bit in each word of the matrix. This can take longer than you want to lock out file system write operations, so you may have to allow writing during cloning; also, the new matrix has to be written to disk.

Limited number of clones There can be at most

32 clones (20 in WAFL).

A more sophisticated data structure may reduce these problems; the following properties can be exploited: most blocks are alive in all clones, or dead in all clones; large extents of blocks will be all alive or all dead everywhere.

If we have per-block liveness information, it may be preferable to simply write to dead blocks instead of using a cleaner to clean segments. WAFL uses this approach, but it is unclear how well it performs without NVRAM support, especially in the presence of frequent fsyncs or syncs. It also requires some sophistication in selecting where to write.

3.4 Write Organization

We have not described how LinLogFS organizes the blocks within a segment, because we intend to change this (the current organization is not very efficient). The basic constraints are

- It must be possible to find all commit blocks written to disk since the last roll-forward start point.
- If a commit block is on disk, all previous commit blocks (starting at the roll-forward start point) are on disk, too.
- If a commit block is on disk, all the data it describes are on disk, too.

Note that there is no dependence between data writes belonging to different commit blocks, or between commit block writes and data writes belonging to later commit blocks.

We experimented with several ATA and SCSI disks (by performing synchronous user-level writes to the partition device), and found some interesting results:

Write caching in the drive can perform writes to disks out-of-order and has to be disabled if we want to satisfy the dependence constraints by write ordering.[5] Without write caching, all disks we measured lose a disk revolution for each consecutive write (tagged command queuing won't help us when we have to wait for the acknowledgment of the data write (and previous commit block write) before starting the current commit block write).

By placing the commit block at some distance (20KB in our experiments) behind the last data block, we can write the commit block in the same revolution and reduce the time until the commit is completed. This is useful in the case of frequent fsyncs. The blocks between the data blocks and the commit block can be used by the next write batch.

4 Current State and Further Work

Much of what we have described above is still unimplemented. LinLogFS currently performs all the usual file system functions except reclaiming space. It provides in-order semantics (even in the presence of fsync()). LinLogFS uses the roll-forward optimization, but does not use any change records yet.

The only component missing to make LinLogFS practically useful is the cleaner. Other features that are still missing and will be implemented in the future are: cloning (including writable, clonable clones), change records for block pointers, pinning segments containing a file down (for LILO), fast update of network mirrors, efficient growing/shrinking/migrating the volume. We are also considering a version of LinLogFS as a low-level OBD driver (see www.lustre.org and Section 8).

5 Layering

5.1 Logging Block Device?

The difference between traditional and log-structured file systems is mainly in block writing. Most other parts of the file system (e.g., dealing with directories or permissions) are hardly affected.

So we first considered realizing the logging functionality as a block device driver. We decided against this, because:

- The block device driver interface is not sufficient to make a conventional file system log-structured without significant changes in the file system (atomicity for multiple block writes, crash recovery).
- A logging block device driver would interfere with other functionality implemented in the block device layer (software RAID).

[5] The disks we measured only wrote out-of-order if the same block was written twice (in some caching window), but there is no guarantee that other disks behave in the same way nor that we have seen all cases of out-of-order writing.

5.2 Logging Layer!

We decided to reuse most of the ext2 code. We defined a logging layer that sits below the rest of the file system. It is also possible to turn other file systems into log-structured file systems by adapting their code to work on top of our logging layer, but we have not done so yet.[6]

This separation into layers proved to be beneficial due to the usual advantages of modularization (e.g., separate testing and debugging).

The adaption of the ext2 filesystem turned out the be relatively straightforward because the ext2 implementation does a good job in encapsulating indirect block handling from the rest of the filesystem implementation. The interface provided by the logging layer is similar to the one offered by the buffer cache/block device driver thereby easing the adaption of the ext2 code.

Furthermore, we have added calls that bracket every filesystem operation to the ext2 filesystem layer. This ensures that dirty blocks do not get flushed to disk while a filesystem-modifying operation is taking place. This approach guarantees that the filesystem is always in a consistent state when a commit to disk is taking place.

6 Implementation Experiences

6.1 The Linux Kernel Framework...

Like many other implementations of Unix-like operating systems, Linux uses a uniform interface within the kernel (the *VFS Layer*) to access all the various filesystems it supports.

Usually local filesystems do not directly write to the block devices they reside on. They make use the *buffer cache*, a unified cache that helps to avoid costly synchronous block device I/O operations in order to fulfill VFS requests.

Instead of doing synchronous block I/O, they rather mark blocks that are in the buffer cache as dirty (i.e., they need to be flushed to the device) when a VFS operation has caused a change in a block. It is then the task of the kupdate kernel thread to actually commit changes to the device asynchronously.

So a local filesystem can be seen as a filter that turns VFS calls into block read/write operations passed to the buffer cache.

The kupdate kernel thread will start writing dirty buffers back to the underlying block devices when one of the following conditions gets true:

- there are dirty blocks that have aged beyond a certain threshold,
- the operating system is running low on memory and needs to free up some RAM,
- there is an explicit request to update the block device, such as a sync() call.

6.2 ... And Changes Made to it

6.2.1 Extending The Buffer Cache

While this approach works fine for traditional file systems, such as ext2, it is not applicable for filesystems that want to ensure certain consistency guarantees for on-disk data structures. In order to be able to make such consistency guarantees, a filesystem implementation needs a more fine-grained control over the actual order in which writes are performed. However, this cannot be ensured by using the traditional buffer caching mechanisms for writes.[7]

LinLogFS also faces another difficulty using this approach. Since write operations are grouped into segments, the actual on-disk location of a block is not known until the block is actually about to be written out to disk. On the other hand, the buffer cache uses the device and the block number in order to locate a cached item.

Therefore, LinLogFS assigns block numbers past the physical end of the underlying device to unwritten dirty blocks. When a block is about to be written out, an *address fixup* is performed on the filesystem's meta data to reflect the final on-disk location. However, this cannot be handled using the buffer cache alone because some additional information is required in order to accomplish that task:

In order to be able to perform the address fixup, it is necessary to know the inode of the filesystem

[6] Since logging changes the on-disk data structure, this would not be useful for file systems that are only used for compatibility, such as the MS-DOS file system. It would be useful for file systems that offer special features, performance or scalability advantages like XFS. We designed the logging layer and the on-disk structures to allow several adapted file systems to work on top of the logging layer at the same time (this is an extension of the cloning functionality).

[7] Of course it would be possible to bypass the buffer cache completely. But this is not desirable due to the performance impacts this approach will have.

Item	Modification
Filesystem Layer (new)	Based on ext2 Handles perms, dirs, etc... Performs address fixup on request
Logging Layer (new)	Lock dirty blocks in memory Trigger off address fixup Trigger off re-hashing of blocks after fixup
Buffer Cache	Added support for re-hashing blocks Prevent random writes Extended *buffer_head* for address fixup
Inode Cache	Added code for locating an inode in the cache

Figure 3: Overview of modifications to the Linux kernel

object the block belongs to and the logical offset within it. However, the Linux 2.2 buffer cache does not preserve this information.

Because of these obstacles, LinLogFS uses a slightly different approach when it comes to dealing with write operations: Instead of leaving the decision when to write which block to disk to the buffer cache, LinLogFS locks dirty blocks in the buffer cache. This prevents them from being flushed to disk by the kupdate daemon at free disposal. LinLogFS performs direct block I/O instead when enough blocks are dirty or a sync has been requested. This also allows the block address fixup to be performed just before the block is actually being written out to disk.

This approach of determining the address of a block upon the disk write could also be used to optimize other file systems, e.g., ext2: When the block is actually written to disk, usually its file is already closed. Therefore the file system knows the file's probable final size, and can allocate the files on the disk in a way that avoids file and free space fragmentation.

6.2.2 Implementing the Address Fixup

In order to be able to perform the address fixup on pending filesystem meta-data before a data block is written out to disk, we need to store some additional information about the block. Currently, this is done by extending the structure that is used to describe entries in the buffer cache *buffer_head* by a few additional members:[8]

[8]We are planning to reduce these additional members in the future. In general, we would suggest adding a general-purpose pointer member to this structure.

- *inum:* This member holds the inode number of the filesystem object this block belongs to.
- *log_blknum:* The offset (logical block number) of this block within the filesystem object it belongs to.
- *private_data:* A pointer to a data structure describing the LinLog filesystem. This eases the porting of existing filesystem implementations to LinLogFS's logging layer since this allows an interface similar to the one of the buffer cache to be used.[9]

When LinLogFS decides to write out dirty blocks the logging layer fixes up the block addresses in the *buffer_head* structs of all blocks that are about to be flushed to disk. It then issues a callback to the upper filesystem layer that is now supposed to adjust its meta-data information accordingly.

Since the buffer cache uses the on-device block number to hash entries, the respective blocks need to be re-hashed in the buffer cache after the address fixup has been performed. Therefore, a function has been added to the Linux buffer cache that allows to rehash blocks in the buffer cache after the address fixup has been performed.

The address fixup is also complicated by the fact that there might actually be two copies of an inode in memory: One stemming from data blocks of the .ifile (the file containing all the inodes in LinLogFS) and another one being located in the inode cache. The inode cache in Linux is mainly used for storing a unified representation of a filesystem inode that has been obtained through the VFS layer. All processes wanting to access a certain file are associated with

[9]This allows us to simulate routines, such as *mark_buffer_dirty* or *bread*.

the representation of the file's inode in the inode cache.

However, the inode cache causes problems for LinLogFS, since it needs to make sure that the inode's copy in the inode cache remains consistent with the one in the .ifile's dirty buffers. Therefore, a call has been added to the Linux inode cache implementation that allows to obtain the current cached copy of an inode in the inode cache (if there is one at all). This allows us to locate a cached copy of an inode and to keep it consistent with the inode's actual state. We are currently considering to bypass the inode cache for filesystem-specific information in future versions of LinLogFS.

After the upper filesystem layer has completed the address fixup, the actual disk write is triggered by the logging layer by issuing an *ll_rw_blk* request to the underlying block device.

7 Performance

We compared several Linux file systems:

ext2 (Linux 2.2.13); this is the standard Linux file system, and represents traditional file systems. It performs all writes asynchronously. Ext2 does not give any consistency guarantees by ordering writes; instead, it relies on a sophisticated fsck program for crash recovery, but successful recovery is not guaranteed.

reiserfs (3.5.19 on Linux 2.2.14); this file system tries to scale well to dealing with large numbers of small files. Concerning recovery guarantees, this file system logs meta-data, so the directory tree will be preserved on crash recovery, but the data in it may not.

ext3 (0.0.2d on Linux 2.2.15pre15); this is the ext2 file system with journaling added; currently it logs both data and meta-data, which allows for very good consistency guarantees.[10].

linlogfs (on Linux 2.2.14); the version we measured did not contain a cleaner and does not keep track of segment usage as a version with a cleaner probably would; there are also several other changes planned that will affect performance. Concerning consistency guarantees, Linlogfs provides the in-order guarantee (see Section 3.2), so crash recovery will restore both data and meta-data to the state a short time before the crash (ext3 probably provides a similar guarantee).

The hardware we have used for this performance comparison is based on a 450MHz AMD K6-III CPU, an Asus P5A mainboard (with ALI chipset), 128MB RAM and a 15GB IDE Harddisk (IBM-DJNA-351520).

For all benchmarks except one we disabled write caching on the hard disk with `hdparm -W0`. This is necessary for reiserfs, ext3, and LinLogFS, because with write caching enabled disks can perform the writes out-of-order in a way that would compromise the crash recovery guarantees these file systems make. Ext2 makes no crash recovery guarantees, but write caching must still be disabled to ensure that ext2 works correctly for `sync` etc. We have also run ext2 with write caching, because this is a common configuration (all IDE hard disks we know enable write caching by default).

Many factors influence file system performance, and it is probably impossible to reduce the wide range of usage patterns in real life to a small set of benchmarks that exercise all file systems in representative ways. So take the following with a grain of salt. The benchmarks we used were:

- Un-tarring the Linux 2.2.14 kernel sources, starting with a freshly-made file system; we ran this benchmark 50 times. Un-tarring is one of the few disk-intensive tasks we encounter in real life. This is a data writing benchmark.
- A single run of removing the 50 instances of the Linux kernel source trees generated by the previous benchmark. This is another one of the few disk-intensive tasks we encounter in real life; this benchmark mainly writes meta-data, in contrast to the un-tar benchmark.
- Tarring[11] the Linux 2.2.14 kernel (20 runs). This is a data reading benchmark.
- A single run of find, searching for "foo" (non-existent) in 20 Linux kernel source trees. This is a meta-data reading benchmark.

In addition to running these benchmarks on the various file systems, we also measured the limits of the hardware: We wrote 74MB (the same amount of

[10] For higher performance at reduced safety meta-data-only logging is planned

[11] We used `tar c linux|cat >/dev/null`, because tar seems to optimize direct output to /dev/null.

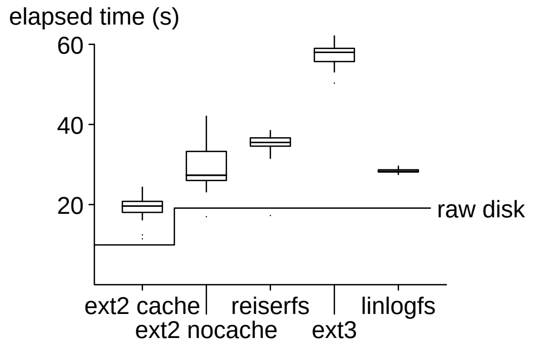

Figure 4: Un-tar timings

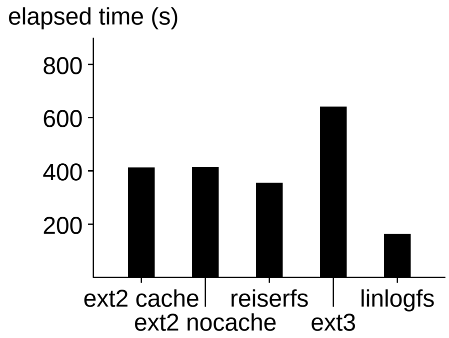

Figure 5: Rm timings

data as written by the un-tar benchmark) to the device with `dd if=/dev/zero of=/dev/hda7 ...` to measure the disk speed limit.

We ran the benchmarks using the default settings for the IDE chipset: no DMA (not supported for this chipset), no multiple sector access, 16-bit data transfers; with write caching disabled, enabling multiple-sector access and 32-bit data transfers make no difference for the elapsed time for direct device accesses (but reduced the CPU time somewhat). We also performed a few experiments with file systems and observed little difference between the two settings; we also believe that using DMA would not make much difference. The CPU load (as reported by `time`) in all the benchmarks except un-tor on ext2 cache was well below 50%, so the PIO probably did not interfere much with the file system.

With write caching enabled, raw writing becomes CPU-bound, and enabling these features reduces the elapsed time for writing 75MB to the device from 9.9s to 6.3s. So the ext2 cache results would be slightly better with the faster settings. Also, the read performance would be better with the faster settings.

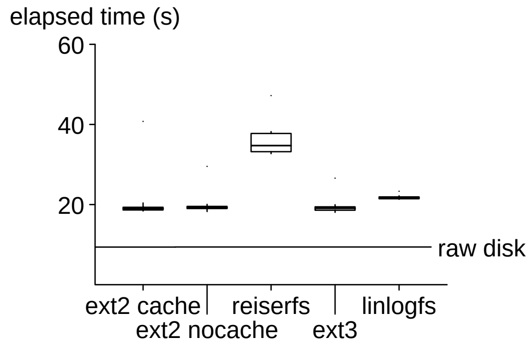

Figure 6: Tar timings

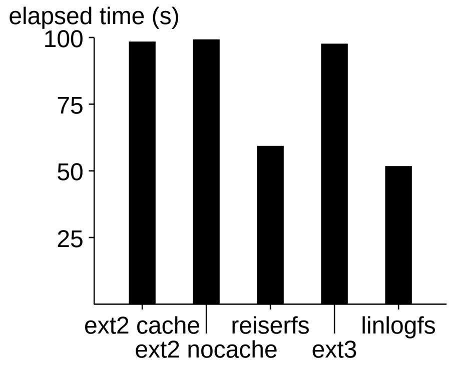

Figure 7: Find timings

The results are shown in Fig. 4, Fig. 5, Fig. 6, and Fig. 7. For the un-tar and tar benchmarks, we display the result as a box-plot showing the median, the quartiles, extreme values, and outliers.

In the local un-tar benchmark, we see that turning off write caching reduces raw disk write performance almost by a factor of 2, but ext2 write performance only by a factor of about 1.5. Ext2 utilizes 2/3 of the write bandwidth of the disk without caching, which is hard to beat. Between ext2, reiserfs and ext3 we see that file systems providing more consistency guarantees perform somewhat slower. LinLogFS breaks this trend, performing as well as ext2 nocache on this benchmark; of course, once cleaning comes into play the performance will suffer. Similarly, the performance of the other file systems on a fuller, aged file system might be worse, too. The outliers below the *raw disk* line that you see for ext2 nocache and reiserfs are probably due to write caching by the file system before memory becomes full.

In the rm benchmark LinLogFS wins, probably because it does not do any bookkeeping of segment usage or free blocks yet; you can expect some slow-down in LinLogFS rm performance in the future.

In the tar and find benchmarks, we see that ext2 cache, ext2 nocache, and ext3 perform the same, because neither caching nor journaling plays a role when reading (i.e., for reading ext3 is the same as ext2).

For the tar benchmark, LinLogFS performs worse than ext2 by about 15%, probably because of the current write organization within a segment. The outliers for most file systems are probably due to interference with the delayed writes of the un-tars that immediately precede the tar benchmark.

For the find timings, LinLogFS and reiserfs are significantly faster than ext2/ext3. The lower performance of ext2 may be due to the way it allocates inodes to different block groups, whereas in a clean LinLogFS data written close together in time is close in space. Our benchmark performs a find on an untarred directory tree; therefore the meta-data written close in time tends to be read close in time, and having them close in space gives an advantage.

The system times reported by `time` show LinLogFS as taking the same or a little less CPU time than the other file systems on all benchmarks; however, take this with a grain of salt, because we saw some strange results in our raw disk measurements, so the reported system time for I/O-intensive benchmarks is not necessarily accurate.

8 Related Work

In many respects LinLogFS is similar to the Sprite LFS [RO92] and especially the BSD LFS [SBMS93]. The most significant difference is that LinLogFS is designed to allow efficient cloning of the file system. One important difference from Sprite LFS is the logical separation of commits and segment summaries; Sprite LFS views segment boundaries as commits and requires additional complexity like the *directory operation log* to work around the resulting problems; BSD LFS logically separates commits from segments (look for *segment batching* in [SBMS93]). BSD LFS does not use change records, Sprite LFS uses the segment summaries as change records for block pointer update optimization.

Another interesting aspect is that the original motivation for log-structured file systems (in particular, Sprite LFS) was performance [OD89], while our motivation is mainly functionality[12] (the performance of ext2 in the usual case is satisfactory). For performance evaluations of log-structured file systems see [SSB+95, MRC+97].

Cleaning heuristics and performance are discussed extensively in [RO92, BHS95].

The Spiralog file system [WBW96] for VMS is a log-structured file system with a number of interesting differences from the file systems discussed above. One of the differences to LinLogFS is its approach to backups [GBD96]: Spiralog just physically backups all live segments; for incremental backups it backs up all live segments written in the corresponding range of time. In contrast, in LinLogFS you will create a logical read-only clone of the file system, and use your favourite logical backup tool (e.g., tar) to make the backup.

Network Appliance's WAFL file system [HLM94] differs from LinLogFS mainly in using free block maps instead of segments and by not performing roll-forward of on-disk structures (i.e., every commit needs a write to the superblock); WAFL is supported by an NVRAM buffer, mitigating the worst-case performance impacts of these decisions. WAFL's allocation scheme trades additional seek times for the elimination of the cleaning overhead; moreover, creating snapshots[13] is significantly more expensive (tens of seconds). Concerning implementation complexity, WAFL needs no cleaner, but probably incurs some additional complexity in the block allocator.

There are two other Linux log-structured file system projects we know of, by Cornelius Cook, and by Adam Richter, but to our knowledge they stalled at some early stage.

Journaling (e.g., ext3 [Twe98]) augments conventional file systems for fast crash recovery by writing data first to a log and later to the home location; after a crash the log is replayed to get a consistent on-disk state. Space in the log is reclaimed by completing writes to the home location. Most journaling file systems (e.g., BFS [Gia99], Calaveras [VGT95], Episode [CAK+92], IBM's JFS, reiserfs) only log meta-data writes (for performance reasons), and therefore only guarantee meta-data consistency. Journaling file systems can provide fast crash recovery and (with data logging) in-order write semantics and relatively good performance for synchronous writes. But they do not easily support cloning,

[12] You can see crash recovery speed as a performance issue, but this is not the kind of performance that most papers focus on.

[13] User-visible snapshots; WAFL also uses the term *snapshot* for commits; commits are not that expensive.

snapshots or other features that are easily available in log-structured file systems. Still, Episode [CAK+92] supports read-only clones, using block-level copy-on-write.

Soft updates [MG99] enhance BSD's FFS by asynchronously writing in an order that ensures that the file system can be safely mounted after a crash without being checked. They make FFS's writes and crash recovery faster. McKusick and Ganger [MG99] also show how to do snapshots in a conventional file system. While they address our main motivations for developing LinLogFS, their software does not work under Linux, and we believe that their solution is more complex to implement than a log-structured file system.

The logical disk [dJKH93] provides an interface that allows a log-structured layer below and an adapted conventional file system on top. The logical disk translates between logical and physical blocks, requiring considerable memory and long startup times. In contrast, our logging layer uses physical block addresses and only affects writes.

The Linux logical volume manager LVM 0.8final supports multiple snapshot logical volumes per original logical volume. This allows making consistent backups for any Linux file system.

The object based disk (OBD) part of the Lustre project (www.lustre.org) divides the file system into a hierarchical layer on top, and a flat layer of files at the bottom, like the layering presented in Section 2. OBD allows inserting *logical object drivers* between these layers. In particular, there is a snapshot driver that uses copy-on-write on the file level. The interface between the OBD layers is higher-level than the interface between the LinLogFS layers. We are considering to have a derivative of LinLogFS as the lower layer in OBD (currently a derivative of ext2fs is used).

9 Conclusion

Log-structured file systems offer a number of benefits; one of the little-known benefits is in-order write semantics, which makes application data safer against OS crashes. LinLogFS is a log-structured file system for Linux. It is implemented by adding a log-structured layer between an adapted ext2 file system and the buffer cache. LinLogFS defers the assignment of block addresses to data blocks until just before the actual device write; it assigns temporary block addresses to dirty data blocks at first, and fixes the addresses just before writing.

The current version of LinLogFS works on Linux 2.2 (versions for Linux 2.0 are available). You can get it through www.complang.tuwien.ac.at/czezatke/lfs.html.

Acknowledgments

Stephen Tweedie (our shepherd), Ulrich Neumerkel, Manfred Brockhaus, and the referees provided helpful comments on earlier versions of this paper.

References

[BHS95] Trevor Blackwell, Jeffrey Harris, and Margo Seltzer. Heuristic cleaning algorithms in log-structured file systems. In *Usenix Annual Technical Conference*, pages 277–288, 1995.

[CAK+92] Sailesh Chutani, Owen T. Anderson, Michael L. Kazar, Bruce W. Leverett, W. Anthony Mason, and Robert N. Sidebotham. The Episode file system. In *Usenix Conference*, pages 43–60, Winter 1992.

[dJKH93] Wiebren de Jonge, M. Frans Kashoek, and Wilson C. Hsieh. Logical disk: A simple new approach to improving file system performance. Technical Report LCS/TR-566, MIT, 1993. A paper on the same topic appeared at SOSP '93.

[GBD96] Russel J. Green, Alasdair C. Baird, and J. Christopher Davies. Designing a fast, on-line backup system for a log-structured file system. *Digital Technical Journal*, 8(2):32–45, 1996.

[Gia99] Dominic Giampaolo. *Practical File System Design.* Morgan Kaufmann, 1999.

[HLM94] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. In *Usenix Conference*, Winter 1994.

[MG99] Marshall Kirk McKusick and Gregory R. Ganger. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem.

In *FREENIX Track, USENIX Annual Technical Conference*, pages 1–17, 1999.

[MRC+97] Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello, Randolph Y. Wang, and Thomas E. Anderson. Improving the performance of log-structured file systems with adaptive methods. In *Sixteenth ACM Symposium on Operating System Principles (SOSP '97)*, 1997.

[OD89] John Ousterhout and Fred Douglis. Beating the I/O bottleneck: A case for log-structured file systems. *Operating Systems Review*, 23(1):11–28, January 1989.

[RO92] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[SBMS93] Margo Seltzer, Keith Bostic, Marshall Kirk McKusick, and Carl Staelin. An implementation of a log-structured file system for UNIX. In *Usenix Conference*, pages 307–326, Winter 1993.

[SSB+95] Margo Seltzer, Keith A. Smith, Hari Balakrishnan, Jacqueline Chang, Sara McMains, and Venkata Padmanabhan. File system logging versus clustering: A performance comparison. In *Usenix Annual Technical Conference*, 1995.

[Twe98] Stephen Tweedie. Journaling the Linux ext2fs filesystem. In *LinuxExpo '98*, 1998.

[VGT95] Uresh Vahalia, Cary G. Gray, and Dennis Ting. Metadata logging in an nfs server. In *Usenix Annual Technical Conference*, pages 265–276, 1995.

[WBW96] Christopher Whitaker, J. Stuart Bayley, and Rod D. W. Widdowson. Design of the server for the Spiralog file system. *Digital Technical Journal*, 8(2):15–31, 1996.