Implementation of a Linux Log-Structured File System with a Garbage Collector

Martin Jambor
Dept. of Software Engineering
Charles University, Prague
jamborm@matfyz.cz

Tomas Hruby
Dept. of Software Engineering
Charles University, Prague
byjac@matfyz.cz

Jan Taus
Dept. of Software Engineering
Charles University, Prague
pan_tau@matfyz.cz

Kuba Krchak
Dept. of Software Engineering
Charles University, Prague
gkg@matfyz.cz

Viliam Holub
Dept. of Software Engineering
Charles University, Prague
holub@dsrg.mff.cuni.cz

ABSTRACT

In many workloads, most write operations performed on a file system modify only a small number of blocks. The log-structured file system was designed for such a workload, additionally with the aim of fast crash recovery and system snapshots. Surprisingly, although implemented for Berkeley Sprite and BSD systems, there was no complete implementation for the current Linux kernel. In this paper, we present a complete implementation of the log-structured file system for the Linux kernel, which includes a user-space garbage collector and additional tools. We evaluate the measurements obtained in several test cases and compare the results with widely-used ext3.

Categories and Subject Descriptors

D.4.3 [**Operating Systems**]: File Systems Management—*file organization*; D.4.2 [**Operating Systems**]: Storage Management—*garbage collection*; E.5 [**Data**]: Files—*organization/structure*

Keywords

Log-structured file systems, Linux file systems, garbage collection

1. INTRODUCTION

As random access memory is getting cheaper and more abundant in both personal computers and servers, many more workloads fit entirely into the disk cache. With most of the read requests satisfied by the cache, it is reasonable to optimize file systems primarily for writing. Moreover, quick crash recovery is a common requirement for a production file system and the most common way of achieving it is *journaling*. On the other hand, this technique incurs a write performance penalty[14] because it needs to write a portion of data twice to achieve a consistent metadata state at any given moment[21].

Log-structured file systems (LFS) have been proposed in [15] and first implemented for Berkeley Sprite system[17] in order to address these two issues. A log-structured file system writes all new information to a sequential structure referred to as the log, thus minimalizing the number of seeks and allowing fast crash recovery. They can also provide additional functionality not easily implemented by traditional file systems, such as snapshots.

Following the Sprite-LFS mentioned above, Seltzer et al.[18] implemented a LFS system for contemporary BSD systems in 1993. Unfortunately, over the course of time it has been removed from FreeBSD and OpenBSD. It is still present in NetBSD but it appears to be no longer completely functional as of NetBSD 2.0.2[22]. There have been several attempts to write LFS for Linux, but all except one have been abandoned without achieving their goals. The only exception is the currently developed NILFS project[11], but it still lacks working garbage collector which is a vital part of any LFS and also the part that poses most implementation issues. In the end of the day, there has not been a working implementation of a traditional[1] LFS for an open-source operating system since 4.4BSD.

In this paper, we present a design and an implementation of LFS for Linux 2.6 kernel which takes full advantage of the page cache, has a working garbage collector, uses sophisticated data structures for large directories that considerably speed up directory operations, implements snapshots and is capable of fast recovery from a system failure. We concentrate on those parts that differ from the BSD implementation[18], how the file system is integrated to the current Linux environment and our solutions to the problems encountered during the implementation of the garbage collector and the segment management in general. We have also done a series of measurements to compare our file system with *ext3*[21].

[1]There are LFS for flash-based devices but they pursue different goals and are not considered by this paper.

We start with an overview of our implementation (Sect. 2), outlining the basic structures and describing the writing process and recovery mechanisms. The most significant issues that have arisen during the implementation are discussed in Sect. 3. This includes the free space and segment management and the garbage collector in particular. We evaluate the file system performance using several benchmarks (Sect. 4). Finally, we discuss related work (Sect. 5) and conclude the paper (Sect. 6).

2. IMPLEMENTATION OVERVIEW

Although the fundamental principle of LFS may seem simple, it presents us with two important issues. The first is a need for an indexing structure so that read requests can be performed without sequentially scanning the disk. We have adopted the traditional approach of representing files and directories with inodes and use the well known mechanism of direct and indirect blocks to quickly locate requested data[20]. We write both structures to log whenever changed.

The trickier problem is how to manage free space so that writes are coalesced into large contiguous bursts. As all other LFS, we have solved this issue by dividing the disk into fixed size segments[17, 18], one megabyte each. When processing a write request, LFS must find an empty segment first. Afterwards, it accepts write requests from the memory management, VFS layer or the garbage collector and keeps writing the given data to the allocated segment as long as the currently written entity fits in. When it does not, we finish the full segment by appending metadata (described below) and allocating and writing a new segment from that point on. Thus, we always write the current segment sequentially from the beginning to the end. The drawback of this approach is that we must copy all live data out of the segment before rewriting it.

LFS systems need to finish the current segment also after they have flushed dirty data during `sync` or `fsync` system calls. Moving on to a new segment would potentially waste a lot of disk space. Therefore, these file systems introduce the so called *partial segments*. Each physical segment consists of one or more partial segments (Fig. 1). Multiple partial segments usually result from the system calls mentioned above. Detailed description can be found in the official documentation of the project[8]. Each partial segment contains the following elements:

1. Data blocks and indirect blocks of files and directories.
2. Inodes
3. Journal enabling the roll forward utility to deal with directory operations (Sect. 2.3).
4. File info structures required to identify all data and indirect blocks so that the garbage collector can recognize live data (Sect. 3) and the roll-forward utility understands which data have been changed since the last checkpoint (Sect. 2.3).
5. Segment summary which contains information global to the partial segment, including but not limited to checksums, addresses and sizes of the individual entity blocks described above, and so on.

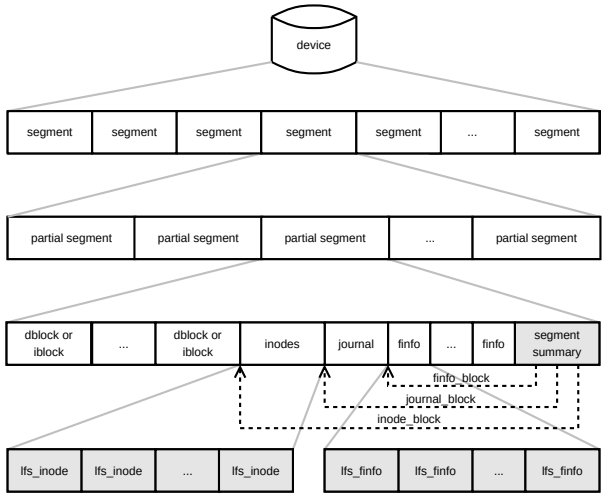

Figure 1: The internal structure of segments and partial segments.

Clearly, LFS must have some means of tracking segment states and inode positions. Sprite-LFS did so by introducing a *segment usage table* which contains, among other information, number of live blocks and inodes left in segments and an *inode table* to store the inode positions. Both were fixed size stand-alone kernel structures written to the log at file system checkpoints. BSD-LFS and our implementation place both of them into an immutable regular file, visible in the file system, called the *ifile*. This approach simplifies the design because the ifile can be handled with less special-case code, in particular it is cached like any other file. Moreover, the ifile can also grow like an ordinary file and therefore it does not impose any limit on the number of inodes in the system.

2.1 Segment Building

All data and metadata except superblocks are written into the log consisting of segments. Therefore, the primary role of the code performing writes is to create the segments and is usually referred to as the segment building subsystem. Our implementation of LFS does not actively seek any data to write, it simply services write requests issued by the memory manager, the user or the garbage collector. In all these cases, the write requests are demands to flush a dirty page, a set of dirty pages or an inode to the disk. The basic algorithm turns out to be fairly simple:

1. Obtain a free segment from the segment management subsystem.
2. Any request to write a data block or an indirect block which fits into the current segment is carried out in the following steps: First, we schedule all necessary journal records to be written when the current partial segment is finished (see Sect. 2.3). Second, we create a new file info structure that describes the given block and plan to write it when the partial segment is about to be closed. Third, we immediately write the block itself to the end of the log. Finally, we update the segment usage table if needed. Even though we queue some meta-

data, the actual blocks are not queued or copied in any way. This property of the segment building subsystem is very helpful in out-of-memory situations because the allocated structures are incomparably smaller than the written block which is about to become clean and thus reclaimable in the page cache.

3. Requests to write inodes are handled in a simpler way. First, the required journal records are also scheduled like in the previous case. The inode is then copied to an extra chunk of memory and is ready to be written before the system moves onto the next partial segment. The segment usage table may also need to be updated.

4. The subsystem checks whether there is enough room for both the blocks and all the planned metadata in the current segment. If any request cannot be safely accommodated in it, the queued inodes, journals, and file info structures are flushed to the disk together with a newly created segment summary record. Finally, we write new information about the current segment to the segment usage table and store new positions of all inodes in this segment in the inode table. The segment has been finished and the subsystem starts again from point 1.

5. A similar course of events takes place when the current partial segment must be closed because of operations like `sync`. The difference is that there is usually a substantial amount of unused space left in the current physical segment. In that case, we start a new partial segment within it.

Sprite-LFS stored the file information records and the segment summary at the end of every partial segment. It required the underlying block device layer and the disk controller to preserve the order of write requests and thus a presence of a segment summary guaranteed the whole partial segment has been written successfully. BSD-LFS did not make such assumptions, used checksums to verify a segment was valid and the authors have therefore decided to put the two structures at the beginning of a partial segment. Even though we also use checksums to verify segment integrity, the algorithm described above dictates that metadata are placed after the data and indirect blocks because the blocks are flushed to the disk before we even know how much metadata there will be, let alone what its structure will be.

We create consistent checkpoints by flushing all dirty data from the page and inode caches and by writing a consistent ifile. Unfortunately, the algorithm described above cannot produce a consistent ifile because the segment usage table is stored within the ifile and any update of it marks a new block dirty. Therefore, we first create consistent ifile partial segments in memory and flush them to disk after reaching a stable state. This is not a problem though, because creating a checkpoint is never a part of memory reclaiming and so we can safely perform substantial allocations.

2.2 Metadata Caching

In order to work effectively, any file system must buffer all data it may access repeatedly within a short period of time. The page cache is the key mechanism to buffer file data in Linux. Because virtually all file systems use it despite storing data in many very different ways, it offers a great level of control over how the data are read and written. On the other hand, traditional Linux block device based file systems use the device node page cache to buffer metadata. That is convenient as long as the metadata stay in the same place on the disk because the device node cache can flush it to the spot where it was read from at any time.

Obviously, such behavior is highly undesirable in LFS. Because directories can use the page cache, new techniques to buffer indirect blocks and inodes had to be investigated. Let us consider the indirect blocks first. NILFS solved a similar problem by introducing a cache of their own. On the contrary, we wanted to unify the code that deals with direct and indirect blocks as much as possible, maximize the utilization of functionality already offered by the Linux kernel and minimize the number of special cases. We have observed that the page cache itself meets our requirements if we can create an extra stream of pages[2] per every file. This solution proved to work very well. In addition, the number of places where different code paths are taken according to the type of the processed block is very small and all of them are only a few lines at most.

Inodes are compulsorily buffered by the inode cache in the Linux kernel and thus there was no question of using something else. On the other hand, the semantics of the inode cache does not perfectly suit LFS either. When the cache decides to free a dirty inode, it asks the corresponding file system to synchronously write it to the disk and discards the memory structure immediately afterwards. Nevertheless, the segment building algorithm described in Sect. 2.1 does not and cannot actually write the inode until the current partial segment is closed. Closing it whenever an inode is written synchronously because of memory reclaiming would bring about unbearable performance overhead. This means that after the inode cache structure is discarded and before finishing the segment, the current version of the inode is present neither in the cache nor on the disk and the address in the inode table is wrong.

In order to avoid this problem, all inodes we plan to write are also added to an *ihash* hash table and kept there until the disk controller signals they were successfully written. If the kernel requests an inode in the critical time window described above, we obtain it from the ihash rather than from the disk. Moreover, the ihash also helps us to avoid writing a single inode several times into one partial segment and unnecessary inode garbage collection.

2.3 Crash Recovery

LFS currently offers two ways of recovering from an unexpected crash. The simpler one is to continue from the last checkpoint, guaranteed to be in an entirely consistent state, and discard any subsequently written data. Moreover, we also provide a roll forward utility to recover as much information as possible even if it was written after a checkpoint. This utility starts with the last checkpoint and follows the chain of segments that have been written since then as long as their checksums are correct. All entities in each of these

[2]This is called a *mapping* in the Linux kernel terminology.

segments are identified by examining the segment summary and file information records and the appropriate metadata are updated. If a data block is read, the utility modifies the relevant indirect block or inode. If an inode is encountered, it updates the inode table so that it points to this copy of the inode. In both cases, segment usage table must also be modified.

The utility must also deal with consistency between directories and inodes. If a crash occurs while only a part of a directory operation has been written to the disk, the link counter of an inode might not match the number of directory entries or such an entry may refer to a non-existent or even a wrong inode. The basic problem is that most directory operations affect multiple inodes and either all changes or none at all must be recovered during roll forward. BSD-LFS fights with the problem by marking all partial segments that contain an unfinished directory operation and not performing roll-forward of them unless they are followed by a segment that completes the operations. Sprite-LFS and our implementation insert a record for each directory operation to the log . These records together form a *directory operation journal*. Both file systems guarantee that the corresponding journal record appears in the log before any affected inode or directory block. Even though this approach makes roll-forward more difficult to implement, it allows recovering more data and imposes very few requirements on the implementation of the directory operations which thus can be simpler and quicker.

2.4 Directories

The log-based organization performs very well in situations where small files are manipulated intensively. Because such workloads update directories frequently, the directory manipulation operations should be performed in logarithmic time. Therefore, for directories larger than one block, we use the *htree* indexing method proposed by Danies Phillips[16] and currently used in ext3. Our implementation follows the paper closely, except that our data structures are simpler because we do not need to provide backward compatibility with early ext2 directories.

3. FREE SPACE MANAGEMENT AND GARBAGE COLLECTION

Free space management in LFS has two main goals. It provides free segments to the segment building subsystem and it recognizes and deals with "out of free space" situations. These situations must be detected by the system call handler because we cannot signal the error to the user space afterwards. Therefore, whenever we allocate a new block or an inode, we appropriately decrement a global free space counter stored in the superblock, unless it would become smaller than a certain threshold value. In that case we return an error to the user space. Conversely, whenever the user deletes an entity, we increment the same counter. The threshold value is set to 15% of the total disk size and is required to store metadata and provide some extra space for garbage collection.

We have already stressed that all live data must be copied out of a segment before reusing it. However, there may be almost no free segments available even though there is enough free space on the disk. Empty segments are created during *garbage collection* by rewriting live data from underutilized segments. There are four important issues when doing so:

1. the file system must be able to detect situations when it is essential or profitable to start cleaning,
2. the best segments to empty must be identified,
3. live data within those segments must be identified, read and appended to the current end of the log, resulting in a smaller number of near-full segments,
4. the selected segments must be reclaimed once it is safe.

We implemented the second step in the user space and the rest of the garbage collector in the kernel. All four steps are nontrivial and we will cover them in the rest of this section.

3.1 Segment Preallocation

The segment building code cannot start garbage collection at the moment it requests a new segment – it is already too late at that point. It could have been called because memory is low and so any memory allocations can either fail or block until the issued write finishes. Since identifying and reading live data from a segment may require a lot of memory allocations, both could cause a deadlock. Moreover, since the decision which segments are to be cleaned is done in the user space, any access to memory can cause a page fault and a blocking memory allocation which would also lead to a deadlock.

To avoid this situation, we track the number of dirty blocks and inodes in the cache. We compare it to the space in currently free segments each time a system call or the page fault handler is about to mark another block dirty. The key idea is to ensure that all dirty blocks can be written to the currently free segments at any time. This method is called *segment preallocation*. Additionally, we always keep a number of free segments for the garbage collector and the ifile. Therefore, when a block is about to become dirty, but there are not enough free segments to satisfy the constraints described in this section, we suspend the current process until enough segments are emptied and activate the garbage collector with an *emergency* message.

This scheme cannot be applied to the ifile because its blocks are regularly marked dirty by the segment building code which cannot wait for garbage collection. Still, we must guarantee a place in free segments for dirty ifile blocks too, otherwise the system might deadlock. Therefore we add the size of the ifile to the mandatory segment reserve. This means that preallocation acts as if the entire ifile was cached in RAM and dirty, even though it may not necessarily be so.

3.2 Segment Selection and Cleaning

Sprite-LFS had all components of the garbage collector incorporated in the operating system kernel. BSD-LFS has moved it to the user space so that different cleaning strategies tailored to different workloads could be easily implemented. Additionally, their garbage collector used the ifile

to learn about segment utilization. Our implementation also makes the selection decisions in the user space for the same reasons. Nevertheless, we returned the rest of the cleaner back to the kernel so that it can efficiently communicate and synchronize with the segment building code through the page cache. The user space and kernel parts communicate by the *NETLINK* protocol[2] so that there is only one communication channel independent of the on-disk format. The user space selection utility listens on the NETLINK socket for information about segment usage changes and cleaning commands. Cleaning commands are issued by the kernel when there is an imminent shortage of segments or the file system has been idle for a given period of time. Even though the kernel can support a number of segment selection utilities, so far we have implemented only one based on the cost-benefit algorithm [17]. When the utility selects a particular segment to clean it sends back a request to the kernel over the same NETLINK interface. We clean the selected segment in the kernel on the behalf and within the context of the selection task.

We process the segment by reading the relevant segment summaries, file info structures, and inodes and identifying live entities like Sprite-LFS and BSD-LFS do. When we find a live block, we read it to the page cache, mark it dirty, and pass the whole inode to the segment building code which writes all dirty pages of that inode to the end of the log. It is important to note that both read and write operations need to lock the corresponding page in the page cache. That means no process can have any page cache locked if blocked during preallocation and waiting for the garbage collector to free more segments. However, these processes originally intended to update the contents of a page and thus must release the appropriate page lock before blocking. Fortunately, the generic write functions in Linux kernel 2.6.17 and later can handle this situation and reacquire the lock if necessary. Flushing all dirty pages and not just those read by the garbage collector is deliberate because it stores adjacent blocks in the file next to each other on the disk. Preallocation guarantees there are free segments for both the blocks that were garbage collected and those modified by the user so this can never exhaust the segment reserve for cleaning.

When all data blocks, indirect blocks, and inodes in a segment have either been deleted by the user or moved to a different place on the disk, the segment becomes empty. But there are important reasons why it cannot be reused immediately. Consider the following situation: the user has modified a data block and thus a write request has been issued to store it to a new location on the disk. At the same time, the segment usage is decremented and drops to zero. If the segment was immediately reclaimed and overwritten, the block subsystem could reorder the writes so that the old copy of the block is overwritten before the new one safely lands on the disk. If a system crash occurs within this time interval the block would be lost. The segment management therefore never reuses a segment unless all segments to which data could be moved have been safely written to disk. Moreover, when roll-forward is turned off, segment reuse must be postponed until the next `sync`. Otherwise, we could not recover from crashes by simply continuing from the last checkpoint on because parts of the consistent checkpoint file system state might be overwritten (Fig. 2).

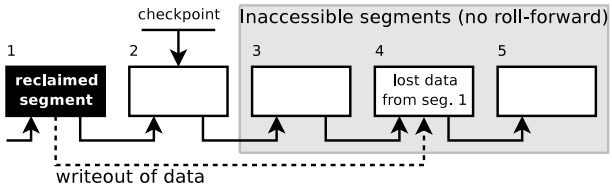

Figure 2: Roll-forward is off: Data from segment 1 were moved. Segment 1 was reclaimed before checkpoint was updated resulting in loss of that data.

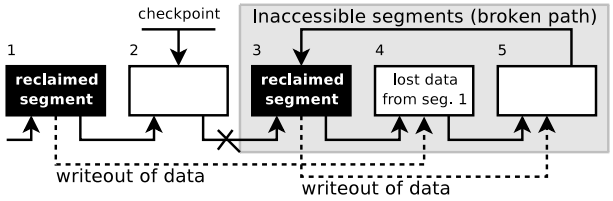

Figure 3: Roll-forward is on: Data from segment 1 and 3 were successfully moved. Segment 1 was reclaimed but this is no problem because data can be reconstructed during the roll-forward. Then segment 3 is reclaimed resulting in loss of data from segment 1 because the path of segments for roll-forward is broken.

3.3 Segment Reclaiming

On the other hand, if roll-forward is enabled, the file system must take care not to reuse any segment that is younger than the current checkpoint. Segments form a singly-linked list which would obviously be broken by overwriting any of its items (Fig. 3).

3.4 Snapshots

A snapshot is a read-only copy of the file system at a particular moment. We have implemented them as special file systems and expect them to be used primarily for online backups. When a snapshot is mounted, the real file system creates a checkpoint. We record the ifile inode position within this checkpoint and use it in the same functions that perform reading of the live data. In this way, the snapshot allows the user to access all data stored in the file system at the time of the snapshot mount. Naturally, the live file system must not overwrite the segments containing these data. To enforce this, the garbage collector does not consider segments that are older than the snapshot checkpoint and never reclaims them until the snapshot is unmounted. We currently support only one snapshot at a time but the scheme can be easily extended to support multiple concurrent ones.

When a snapshot is mounted, the free space management of the live file system behaves somehow non-intuitively. Naturally, every block which has been modified after the snapshot was mounted must be stored twice, thus occupying twice as much space on the disk. The disk free space therefore decreases even by modification of already existing data, not only when new are created. Most intriguingly, the user may run out of free space by simple file deletion because there is no space left to store the modified copy of the directory.

Processor	AMD Athlon XP 2500+ family 6 model 10 1338MHz
Chipset	VIA KT400
Memory	256MB DDR333 SDRAM (Single Channel)
System disk	120GB Western Digital, Serial ATA
Measured disk	Seagate Barracuda ST380817AS, Serial ATA 80GB 7200.7 Firmware 3.42

Table 1: Reference computer configuration

As stated above, over the lifetime of the snapshot, we do not reuse segments that were not free during mounting. That means the life system has only the free segments at its disposal during this time. Therefore, when the snapshot is mounted, we set the new free space to 85% of the free segments size. Conversely, when the snapshot is unmounted later on, we mark all segments which have meanwhile become empty as immediately usable. We also calculate the new free space by taking its value before the snapshot was mounted, subtracting the space consumed and adding the size of data that were deleted or modified while the snapshot was mounted.

4. EVALUATION

We have carried out a number of measurements[8] to evaluate the performance of our LFS implementation under different workloads and compare it to the most common Linux file system today, namely ext3. We ran all tests described in this section multiple times. The variance between corresponding values was insignificant (below 5%) in all tests. All results together with the exact options used to obtain them are available on our web site[8], this section covers the more interesting ones.

The measured file systems on the reference computer (Tab. 1) always resided on a 80GB Seagate Barracuda drive while the rest of the system was placed on a 120GB Western Digital. The memory available was intentionally fairly low so that data sets bigger than the available cache did not have to be huge.

4.1 IOzone Benchmark

IOzone[4] is a filesystem benchmark capable of measuring a wide variety of file system operations. We have used it to run four series of measurements. Two of them were single-threaded and two involved eight threads. Additionally, two included the time taken by an `fsync` after writing, unlike the other two. In each series, different record and file sizes were used. The record size is the request size issued by IOzone. Each thread manipulates a separate file of the given size and reads or writes this amount of data in each test. The amount of data transfered in multi-threaded benchmarks is therefore eight times bigger than in the single-threaded ones. The maximum file size was always 512 megabytes which was twice the amount of available RAM. As we expected, the performance dramatically dropped when the working set was bigger than the memory available for the disk cache in most of the tests. As you can see in the graphs in this section, this occurs with file sizes about 256 megabytes in the single-threaded cases and 32 megabytes in the multi-threaded ones.

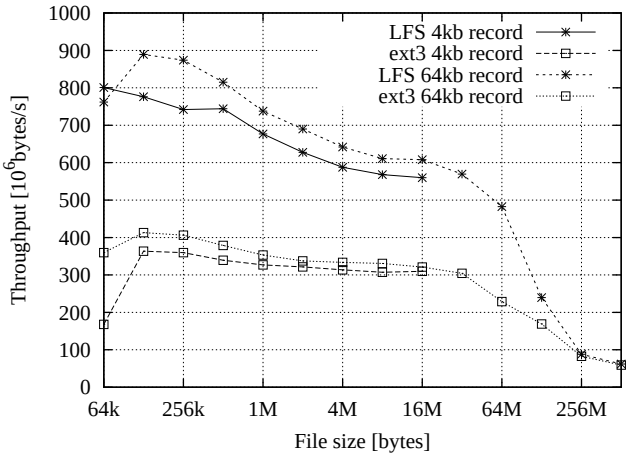

Figure 4: Single-threaded new file creation

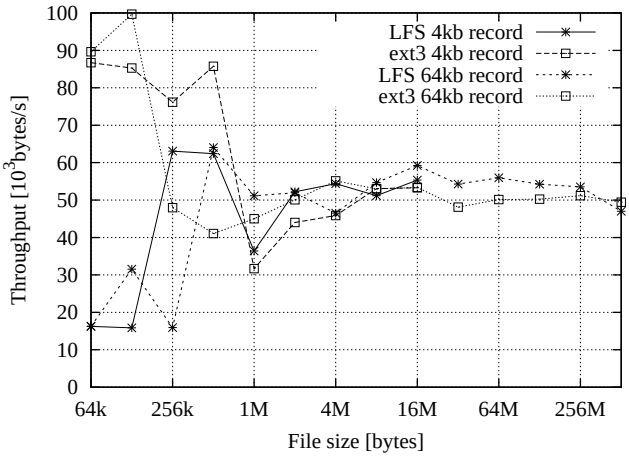

Figure 5: Single-threaded rewriting with fsync

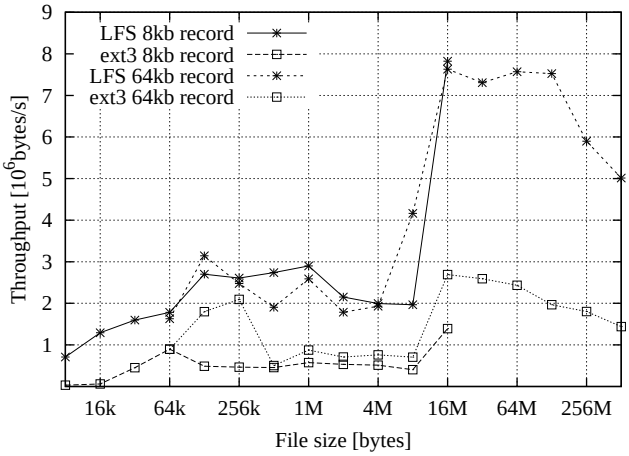

Figure 6: Multi-threaded random write with fsync

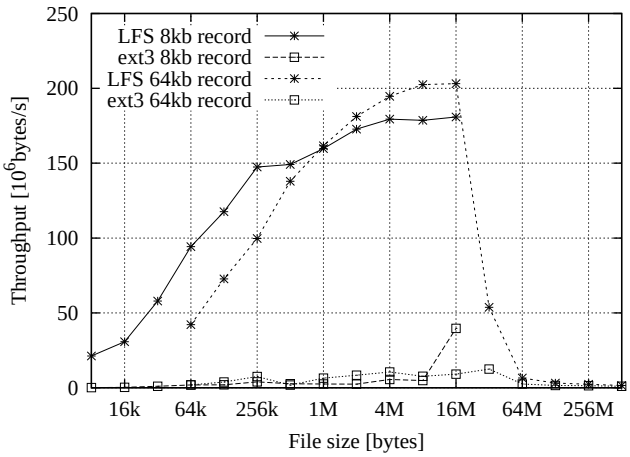

Figure 7: Multi-threaded mixed workload with fsync

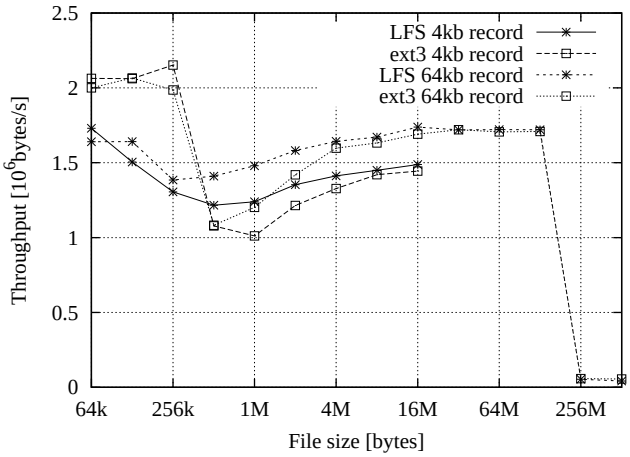

Figure 8: Single-threaded file read

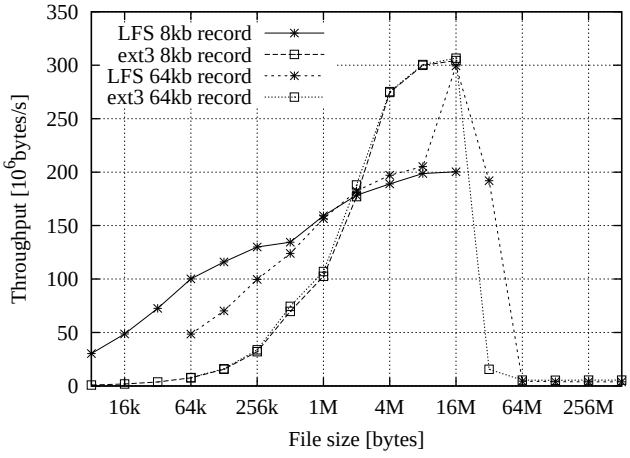

Figure 9: Multi-threaded file read

LFS performed extremely well in creation of new small files in both runs without a subsequent `fsync`. This was expected because ext3 needed to read and process metadata from the disk while LFS did all processing in memory. We have even observed the effect of CPU cache in the single threaded test (Fig. 4) in the case of the smallest files because the amount of transfered data was very small. Nevertheless, results described in section 4.2 suggest that LFS is twice as quick as ext3 even when creating large numbers of such small files. The speed of creating files of the same size as the amount of RAM or bigger is comparable to ext3 because both file systems need to evict data from the page cache and ext3 also writes it almost sequentially since the data is new. On the other hand, frequent `fsync` operations mean a lot of expensive partial segment finishing and so when they were included in measurements, both file systems have performed comparably. Both file systems also produced similar results when rewriting existing files in all runs except the single-threaded one with `fsync` in which LFS was not as fast at writing small files (Fig. 5), again because of the high overhead of partial segment finishing.

LFS was better in experiments where records were overwritten randomly and immediately flushed to disk by an `fsync`, especially in the multi-threaded case (Fig. 6). LFS has also clearly outperformed ext3 in a *mixed workload with fsync* test in which each thread runs either a read or a write test on a round robin basis (Fig. 7).

Single-threaded read performance of LFS was comparable to ext3 for all but the smallest files (Fig. 8). We believe this is because ext3 implementation is more optimized rather than because of the disk layout. Quite surprisingly, the opposite is true in the multi-threaded case where LFS is better at reading small files but worse at mid-sized ones (Fig. 9).

4.2 File creation and removal

LFS are known to perform very well in metadata dominated workloads [17, 19] such as creating and deleting files. In order to examine the performance of our implementation in this field, we have measured the time required by LFS and ext3 to create and delete half a million files of various sizes, including a final `sync`. The results presented in Fig. 10 show that LFS performs better, often significantly, at creating small files, except for those having 16 kilobytes. We believe ext3 performs unexpectedly well in this particular case due to such factors as cache alignment. LFS is also substantially better at deleting files, particularly mid-sized and large ones. This experiment also proved that our directory operations implementation is efficient.

4.3 Postmark Benchmark

Postmark[10] is a widely used benchmark to assess system performance under small file and generally metadata intensive workloads. It works by creating a number of small files and subsequently modifying them in so called transactions. Each transaction consists of a pair of create-or-delete and read-or-append operations. We evaluated LFS using five million postmark transactions on ten thousand files and default values of other configuration options. Postmark reported LFS was more than seven times as quick as ext3 in all measured operations. This correlates with the good results obtained in the mixed workload of the IOzone benchmark.

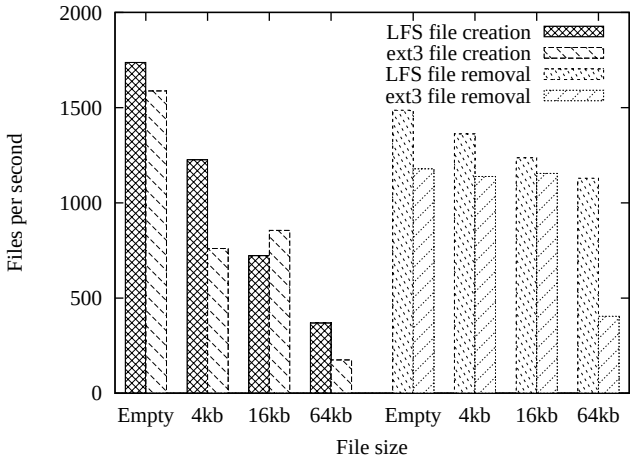

Figure 10: File creation and removal performance

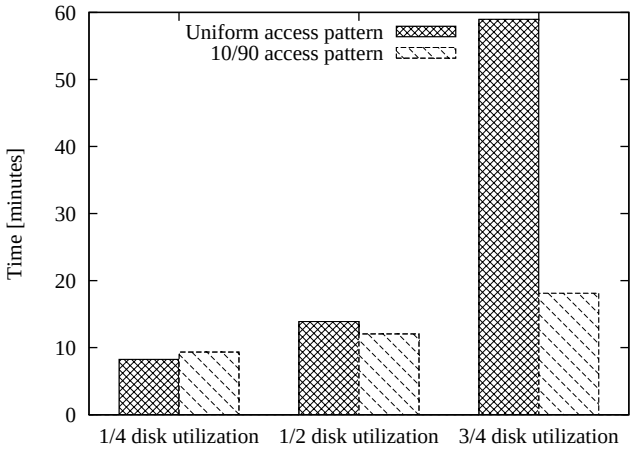

Figure 11: Garbage collector overhead

4.4 Comparison with ReiserFS

We have also done a number of measurements to asses performance of LFS against ReiserFS[3], particularly in the areas where LFS is clearly better than ext3, because ReiserFS was also designed for workloads manipulating small files. In the file creation and removal benchmark described in section 4.2, ReiserFS was clearly better than LFS at creating 16kb and 64kb files and delivered almost exactly the same performance when creating empty and 4kb files and deleting all file sizes. ReiserFS seems to be better in multi-threaded IOzone tests except for the mixed workload one in which LFS is a lot faster and the random write test in which ReiserFS outperformed LFS on files smaller than 8 megabytes. LFS was faster on files larger than 16 megabytes. Last but not least, ReiserFS performed worse than ext3 in the Postmark benchmark (Sect. 4.3). Given the high number of transactions, this is consistent with previous measurements presented in [3]. LFS is fifteen times better than ReiserFS in this benchmark. More detailed comparison with ReiserFS is a subject to future work.

4.5 Garbage Collector Overhead

All measurements described in this section so far involved very little or no garbage collecting at all. However, the need to reclaim underutilized segments is a major drawback of the LFS concept. In order to asses the effect of garbage collection on write performance, we have carried out the following set of experiments. We filled a quarter, a half, and three quarters of a 10 gigabyte partition with data and then measured how much time it takes to randomly rewrite it with 8 gigabytes of data. Rewritten records had 64 kilobytes and were rewritten either with the same probability or with so called 10/90 access pattern in which 10% of hot records are overwritten with 90% probability and 90% of cold records are modified with the probability of 10%. The results can be found in Fig. 11. When the 10/90 access pattern was used, the time required to complete the test when three quarters of the disk were utilized was nearly double of that when only a quarter was. On the other hand, when all records were selected with the same probability, it was over seven times as big. LFS can therefore deliver the performance presented in the previous measurements only if a substantial part of the disk is left unused. Nevertheless, the required amount depends heavily on the access pattern of a particular workload and is quite reasonable when hot data take only a small portion of disk. Moreover, the number of half-empty file systems is big[6] and the file system also carries out garbage collection in periods of inactivity which reduces the overhead when high performance is required.

[3]As suggested by the reviewer. Unfortunately, we did not have enough time to present the graphs in this paper.

5. RELATED WORK

Log-structured file systems have been proposed by Ousterhout and Douglis[15] and implemented for Berkley Sprite system by Rosenblum and Ousterhout[17] in order to reduce disk seeks by collecting large amounts of new data in cache and write them to disk in a single large I/O operation. They also proposed a simple cost-benefit garbage collecting algorithm to reduce the cleaning overhead. Seltzer et al.[18] redesigned several key components such as free space accounting to come up with BSD-LFS implementation that was more robust and better integrated into the contemporary vnode environment. Our work was largely based on these two implementations and the features we share or in which we differ are stressed throughout this paper.

Seltzer et al.[19] extensively compared the performance of the Sprite-LFS[17], BSD-LFS[18] and contemporary *Fast File System*[13]. Blackwell et al.[5] have proposed a heuristic cleaning algorithm to be performed in idle time gaps, based on real workloads simulations. We based our selection of segment size on results presented by Matthews et al.[12] who also proposed adaptive cleaning policies including hole plugging, using cached data to lower cleaning overhead, and reorganizing data to match changing read patterns. Unfortunately, we believe it is difficult to combine these adaptive methods with snapshots.

Digital's Spiralog[9] is a distributed scalable log-structured file system for OpenVMS Alpha capable of online backups. HP AutoRAID[23] adopts log-structured techniques to decrease parity recalculation overhead when modifying existing data by writing them to a new location. Furthermore, a

number of file systems designed specifically for flash EEPROM devices such as JFFS[24] and YAFFS[1] also exploit log-structured format to provide wear-leveling and data consistency. Finally, file systems like WAFL[7] utilize automatic snapshots to allow users to access several past versions of stored data. We provide on-demand snapshots only that represent the state of the file system at the given moment.

6. CONCLUSION

We have introduced a novel and complete implementation of LFS for the Linux kernel with an associated user space garbage collector.

The obtained results show that our implementation outperforms ext3 in workloads with prevailing writes, especially when modifying a lot of metadata, but also when writing ordinary data randomly. Moreover, the speed of read operations is generally comparable to ext3. This performance may degrade because of garbage collection overhead, but we have shown this negative effect is small when the working set itself is small compared to the free space available.

Thus, LFS is suitable for systems such as news or mail servers. Moreover, as disks are continuously getting bigger and their seek times do not improve as much, the advantages of this layout will probably increase while the need for extra space is likely to present an ever smaller problem in the future. Finally, because our implementation is open source, other programmers and researchers may extend on the idea of LFS.

Acknowledgments

The authors are grateful to Lubos Bulej and the anonymous reviewer for their valuable comments. This work was partially supported by the Grant Agency of the Czech Republic project 201/06/0770.

7. REFERENCES

[1] Aleph One Ltd. YAFFS: Yet another flash file system. http://www.aleph1.co.uk/yaffs/.

[2] Linux Netlink as an IP services protocol. RFC 3549, Jul 2003.

[3] Sun microsystems. File system performance: The Solaris OS, UFS, Linux ext3, and ReiserFS, Aug 2004.

[4] IOzone filesystem benchmark. http://www.iozone.org/, Oct 2006.

[5] T. Blackwell, J. Harris, and M. I. Seltzer. Heuristic cleaning algorithms in log-structured file systems. In *USENIX Winter*, pages 277–288, Jan 1995.

[6] J. R. Douceur and W. J. Bolosky. A large-scale study of file-system contents. In *SIGMETRICS '99: Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 59–70, New York, NY, USA, 1999. ACM Press.

[7] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 235–246, San Fransisco, CA, USA, Jan 1994.

[8] T. Hrubý, M. Jambor, J. Tauš, and K. Krchák. Log-structured file system web site. http://dsrg.mff.cuni.cz/lfs/, Nov 2006.

[9] J. E. Johnson and W. A. Laing. Overview of the Spiralog file system. *Digital Technical Journal of Digital Equipment Corporation*, 8(2):5–14, 1996.

[10] J. Katcher. PostMark: A new file system benchmark. Technical Report TR3022, Network Appliance, 1997.

[11] R. Konishi, Y. Amagai, K. Sato, H. Hifumi, S. Kihara, and S. Moriai. The Linux implementation of a log-structured file system. *SIGOPS Oper. Syst. Rev.*, 40(3):102–107, 2006.

[12] J. N. Matthews, D. Roselli, A. M. Costello, R. Y. Wang, and T. E. Anderson. Improving the performance of log-structured file systems with adaptive methods. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 238–251, New York, NY, USA, 1997. ACM Press.

[13] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, 1984.

[14] Namesys. File system benchmarks. http://www.namesys.com/benchmarks.html.

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

[16] D. R. Phillips. A directory index for ext2. In *Proceedings of the Ottawa Linux Symposium*, pages 425–439, 2002.

[17] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. In *SOSP '91: Proceedings of the thirteenth ACM symposium on Operating systems principles*, pages 1–15, New York, NY, USA, 1991. ACM Press.

[18] M. I. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin. An implementation of a log-structured file system for UNIX. In *USENIX Winter*, pages 307–326, 1993.

[19] M. I. Seltzer, K. A. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. N. Padmanabhan. File system logging versus clustering: A performance comparison. In *USENIX Winter*, pages 249–264, 1995.

[20] A. S. Tanenbaum and A. S. Woodhull. *Operating systems (2nd ed.): Design and implementation*. Prentice-Hall, Inc., 1997.

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

[22] wikipedia.org. Log-structured file system. http://en.wikipedia.org/wiki/Log-structured_File_System, Oct 2006.

[23] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems (TOCS)*, 14(1):108–136, 1996.

[24] D. Woodhouse. JFFS: The journalling flash file system. In *Proceedings of the Ottawa Linux Symposium*, 2001.