Revisiting Log-Structured File Systems for Low-Power Portable Storage

Andreas Weissel, Holger Scherl, Philipp Janda
University of Erlangen
{weissel,scherl,janda}@cs.fau.de

Frank Bellosa
University of Karlsruhe
bellosa@ira.uka.de

Abstract -- **In this work we investigate the implications on the energy consumption of different popular file systems and propose a novel, log-structured file system aiming at minimized energy consumption by avoiding expensive disk seeks and introduced latencies due to rotational delays. We show that the energy efficiency of file systems is heavily influenced by the underlying data layout and file organization. Guidelines for a low power file system design are developed and evaluated with measurements of the energy consumption of a prototype implementation. As on-going work we investigate different approaches to free space management. We discuss design choices for the implementation of a family of free space managers and their implications on energy consumption.**

I. Introduction

Energy consumption is an important design issue for modern mobile devices. On the hardware level, this issue is addressed by introducing additional operating modes with reduced power consumption, e.g. hard disks with low-power idle and standby modes. However, research in the area of energy-aware systems has not yet addressed the organization and layout of file systems for hard disks.

The idea of log-structured file systems is to simulate an indefinite storage space (the *log*) to which new or updated data is simply appended. We investigate the potential of this design to minimize disk seeks and rotational latencies and, as a consequence, the energy consumption induced by the file system layout on disk. We present guidelines for the design of a low power file system and evaluate the resulting energy savings with real power measurements of a prototype implementation.

To realize the abstraction of an indefinite log, the available free space on the hard disk has to be managed to reclaim space from old (deleted or changed) data and to hide the physical layout of the hard disk. We discuss possible implementations of free space management and their implications on the energy consumption. Two approaches can be distinguished: in the *copying approach*, a cleaner process is invoked as soon as the length of the log exceeds a certain threshold, e.g. 90% of the size of the partition. This cleaner compresses the log by copying data from the head or the tail into holes (obsoleted data) inside the log. In the *threading approach*, the log is not necessarily stored contiguously but threaded through the holes. If the free space becomes too fragmented, a cleaner process has to compress the log, similar to the copying approach.

Several design choices lead to a family of free space managers which allow the selection of an appropriate algorithm depending on the available energy, the amount of free space which has to be generated and the overhead during normal operation. As work-in-progress, we are currently implementing different free space managers.

The rest of this paper is organized as follows. In the next section, we discuss related work. Guidelines for energy efficient file system design are presented in section III, followed by a presentation of our prototype implementation and its evaluation. In section VI, we discuss approaches to free space management.

II. Related Work

The principles of log-structured file systems (LFS) were presented in 1991 by Rosenblum et al. [3]. A new technique for disk management is proposed that aims at speeding up both disk writes and recovery times and a prototype implementation for the Sprite operating system is presented. Seltzer et al. [6] completely redesigned the file system to integrate it into the 4.4BSD Unix operating system. A performance comparison with the Berkely Fast Filesystem (FFS) showed that log-structured and clustered file systems each have regions of performance dominance [7]. However, they attest log-structured file systems improved performance for reading and writing small files and for their creation and deletion. FFS showed to be more effective when file sizes rise above 256 kilobytes. Czezatke et al. implemented a log-structured file system in Linux by adding a logging-layer between an adapted version of the ext2 file system and the block device [1]. Matthews et al. describe possibilities to improve the performance of log-structured file systems with adaptive methods for a wider range of workloads [2]. Layout adaptation to match hardware characteristics of storage devices through track-aligned extents is performed by Schindler et al. [5]. By utilizing disk-specific knowledge it is possible to increase I/O efficiency by reducing head switches for medium to large requests. Matching segments to track boundaries is suggested to achieve a better performance both for cleaning and writing in log-structured file systems. In [8], file system alternatives for mobile storage and their implications on the energy consumption of different storage media are discussed. Different variants of the traditional UNIX file system are compared with LFS by Rosenblum et al. [3].

III. Guidelines for Energy Efficient File System Design

In the following sections, we discuss requirements and guidelines for the design of a file system for portable storage optimized for low power consumption.

A. Sequential Arrangement of Meta Data and Data Blocks

When writing or reading files from disk, two entities have to be accessed: meta data and file contents. If meta data and file data are stored at physically distinct disk locations, it is necessary to issue two disk requests accompanied by two seeks and two rotational latencies when file data is accessed. Thus, meta data information and file contents should be grouped together to avoid switching between different locations on the disk. Unfortunately, as opposed to a fixed positioning dynamic arrangement of meta data structures introduces complexities and overheads for maintaining their locations on disk. But a careful implementation is able to limit these effects in order to gain the outlined benefits.

B. Free Space Management

Sequential arrangement of file contents and their meta data can be achieved with the abstraction of an infinite storage medium or log. In this case, new or updated files are simply appended to the end of the log. However, data which is no longer valid after updating or deletion remains at its initial position, creating *holes* in the log. To reuse the space occupied by obsoleted data, as described above, the copying or threading method can be used. Free space is generated by compressing the log or by threading the log through extents of old data.

C. Sequential Reading and Writing Behavior

If the file system is fragmented, file access performance will suffer from an imperfect data layout resulting in higher energy consumption. As a consequence, the disk manager should be capable of automatically creating unfragmented empty space on disk. With this process, file data can be rearranged in a sequential manner. However, the task of free space management requires additional energy which has to be taken into consideration.

D. File System Reorganization

Small files that are frequently accessed in the same sequence could also be arranged sequentially to save energy. As described above, data reorganization is already used to generate large extents of empty space. This technique can be extended to match file organization on disk to the access pattern of read operations. This approach is most beneficial for workloads which access many small files. Then additional energy savings are possible by avoiding disk seeks and rotational latency delays between file accesses. An additional policy would be to group files which are residing in the same directory.

E. Adaptive Energy Use

When attached to an external power supply the energy consumption is mostly of no concern for mobile devices. However, when running on battery power, every Joule is precious. Therefore, we distinguish two operating modes: battery and on-line mode. In battery mode, time- and energy-consuming data reorganizations should be avoided and postponed until the device is in on-line mode. Depending on the remaining battery lifetime different cleaning strategies can be used to minimize the additional energy consumption of reorganization while still creating new space. For cleaning policies, we can identify a trade-off between optimizing data and file layout or the amount of free space created and minimizing energy consumption.

F. Fast Crash Recovery

Mobile computers are often used until the battery power is consumed completely leading to unclean shutdowns. In this case or in the presence of software or hardware failures, file systems can be left in an inconsistent state. Therefore, consistency checks have to be performed before accessing the data on the disk to ensure data reliability. Log-structured file systems have the advantage that a consistency check normally does not need to scan over the entire disk as recent changes to the file system can be found at the head or end of the log.

IV. Implementation

From these requirements, the design of a file system based on the log-structured approach emerges. We implemented our prototype file system *scherlfs* (a student thesis [4]) as a Linux kernel module.

A. File System Design

In scherlfs, inodes are not located at fixed disk locations but are stored dynamically in the log. Therefore, a mechanism for translating an inode number to the corresponding disk block is needed. This is done by a simple map lookup operation. Similar to the approach used in LFS, the map is indexed by the inode number and reveals the corresponding logical disk block at this position. Inode numbers which do not correspond to an existing file are associated with a logical block number of zero. The elements of the inode map are placed in a read-only regular file, called the *.ifile*. It is made visible in the root directory of the file system. This approach has several advantages. First, it overcomes the limitation of storing only a fixed amount of files in the file system since data can be easily appended to it as for any other file. Second, in most cases it can be treated as any other file minimizing the special purpose code in the file system. Finally, it is possible to access the file system via the standard system calls when cleaning policies are going to be implemented in user space.

When creating a new inode the .ifile has to be searched for a free entry. Without applied optimizations a linear search has to be performed. In scherlfs, this task is speeded up by the introduction of a free block bitmap which is stored in the superblock. It marks blocks full of inode entries by setting the corresponding bit in the bitmap. At initialization time the bitmap is simply filled with zeroes assuming that all blocks of the bitmap still have free space for an inode entry. Then the entries are updated by a lazy initialization scheme, thus eliminating the need for maintaining the consistency of the bitmap between file system mounts.

The integration of the inode map in the log eliminates the need for a special checkpoint region. When checkpoints are applied, changed .ifile blocks have to be saved to disk and the new position of the .ifiles' inode is simply inserted into the superblock of the file system. This way, the bootstrap problem of retrieving the position of the .ifile by accessing the inode map in the .ifile is avoided.

B. *The Case for a New Inode Design*

The inode of a file has to store all information which is needed to map file block numbers to logical block numbers. As data blocks are not necessarily adjacent to each other most file systems provide a method to store the connection between each file block number and the corresponding logical block number. This mapping goes back to early versions of Unix from AT&T and is also used in LFS [3]. In this scheme inodes store a number of direct and indirect block pointers. However, indirection can cause additional seeks and rotational latencies. Therefore, we decided not to use this approach.

Whenever possible, dirty data blocks of a file are written to disk in large extents. Extents are triples that describe contiguous chunks of data by containing a starting file block number, the corresponding logical block number on disk and length information. By introducing this data structure it is possible to describe even large files with only few extents. This is implemented by saving the extents of a file in a doubly linked list which is added to the in-memory representation of the inode. This way, it is possible to save the file offset only implicitly. An array representation of this list is added to the inode block on disk. Fortunately, for most files only a few extents have to be created. These extents fit easily into the inode structure and can be held in memory together with other inode data. Additional disk seeks are not required in contrast to the traditional representation. However, for highly fragmented files it is possible that a disk inode propagates over a variable amount of contiguous disk blocks. Although additional disk seeks are not required in this situation, the compact representation of the file contents will get lost. Therefore, extent structures have to be carefully managed by the implementation. When updating or creating new extents a more compact representation can be achieved by joining adjacent extents. The opposite case can also occur if extents have to be split into two or three parts.

One major drawback of this scheme appears with applications that issue many small write requests (most Unix utilities issue write calls with a buffer size that matches the system page size). In this case, inode blocks followed by a small amount of data blocks are written out frequently, causing an imperfect data layout on disk by fragmenting the file.

To overcome this problem, scherlfs writes new inode blocks over the disk location of the last written one. To reduce the overhead of additional disk seeks, the inode write operation is delayed a certain amount of time. Deferred updates are already performed by the Linux bdflush (version 2.4) or pdflush (2.6) demon. In scherlfs, the determination of the block number of updated inodes is delayed as long as possible, until the flush demon writes out the inode. This way, reliability is not reduced compared to the already existing Linux implementation.

V. Evaluation

All experiments were performed on a 2.5" IBM Travelstar 15GN hard disk (IC25N010ATDA04, now Hitachi). We determined the energy consumption by measuring the voltage drop at a precision resistor in the 5V power supply line of the hard disk. The voltage drop is acquired using an A/D-converter with a resolution of 256 steps at a rate of 10000 samples per second.

In the area of mobile devices, available memory for caching of file data is limited. To simulate this setting on a desktop PC with plenty of main memory we modified the cache subsystem of Linux to minimize the use of the disk caches: When searching the block buffer cache, the `Uptodate` flag of the requested buffer block is always cleared. Accesses to the page cache reset the `ClearPageUptodate` flag if the requested page differs from the last requested one, thus reducing the page cache to the size of one page. In addition to that the functions `block_read_full_page()` and `block_prepare_write()`, which are used by all tested file systems, are modified to read in also up-to-date buffers.

A. Energy Consumption of Sequential Operations

The first comparative measurements examine the energy consumption of sequential read and write operations across a range of different file sizes. A test program transfers a volume of 50 MB, decomposed into the appropriate number of files for the file size being tested. In the case of small files directory lookup operations dominate all other processing overhead. Therefore, the files are divided into subdirectories, each containing no more than 200 files. Six phases are tested:

- **Create**: The files are created by issuing one I/O operation.
- **Read**: All files are read in their creation order with one I/O operation.
- **Rand_read**: All files are read in pseudo random order with one I/O operation.
- **Rewrite**: All files are truncated and rewritten in their creation order; their original sizes remain unchanged.
- **Reread**: The read phase above is executed again on the file system after the rewrite phase.
- **Rand_reread**: The rand_read phase is executed again on the file system after the rewrite phase.

Traces of file system accesses are recorded and replayed without idle periods on the file systems reiserfs, fat32, ext2, ext3 (mounted in default mode *ordered*) and scherlfs. The energy consumption of the test runs is shown in figure 1. Only the results for a file size of 4 KB are shown, tests for other file sizes show the same trends. Every test run is repeated at least five times; the figures show the average energy consumption. For the majority of tests the variance is negligible, otherwise it is shown in the figures. While energy savings are mainly due to increased performance (the time it takes to replay the trace files on the different file systems mimic the energy consumption shown in the figures), additional savings are possible as the idle periods are increased and there are more opportunities to switch to standby mode.

In the *create* phase reiserfs proves to be most energy efficient in comparison to the traditional file system layouts. Fat32 is very close to reiserfs but cannot be better for any of the different I/O sizes. Ext2 is far behind the other file systems. The design based on block groups shows to be inefficient. The ext3 file system behaves even worse because its recovering technique introduces additional overheads. The log approach of scherlfs proves to be optimal, reaching the highest energy savings for all file sizes. Relative to reiserfs energy savings of 66% to 79.9% are achieved for the creation of small files. Although the relative savings decrease with larger file sizes energy efficiency is still increased by 37.4% for 1024 KB files. This becomes possible because all file data and meta data can be written to the log in a sequential manner. Disk seeks are only necessary for directory updates.

The file systems show only minor differences in their sequential *read* performance. For small file sizes the retrieval of file information from the directories influences the energy consumption significantly. For that reason reiserfs has the smallest energy requirements. The good performance for 4 KB files can be explained by the tail packing of reiserfs. Small files are stored directly in reiserfs' search tree. Therefore, no additional seeks are required when the file object is found in the tree. Although scherlfs arranges all files sequentially in the log and the read order of the files is not varied, the directory lookups have a negative impact on energy consumption.

However, in the *random read* phase the file arrangement of scherlfs shows to be energy efficient because file meta data and file data is arranged sequentially on disk. Fat32 stores meta information for the files together in the directories and in its file allocation table, resulting energy requirements similar to scherlfs. It has to be noticed that this is due to the internal FAT cache of fat32. For larger file sizes the file systems have only minor differences in their energy consumption.

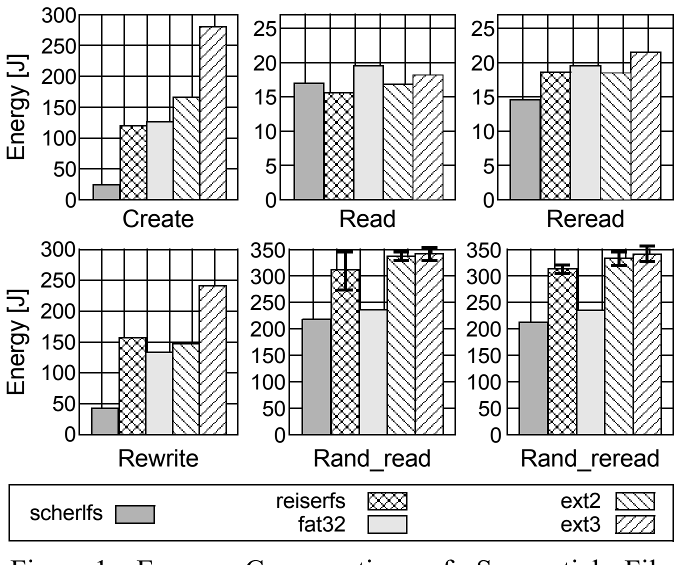

Figure 1: Energy Consumption of Sequential File Operations (4K files)

During the *rewrite* phase, the log approach achieves the highest savings of all file systems, at least for small files. Ext2 and ext3 have both very high energy requirements for all file sizes because of their frequent meta data updates.

After the rewrite phase the two read tests are performed again on the updated file systems (*reread* and *rand_reread*). The energy consumption of the traditional file systems increases a bit. This can be explained by the truncate operation of the rewrite phase which forces the file systems to reallocate the data blocks to the files in contrast to simply overwriting them. The previous allocation scheme can be changed which possibly fragments meta data and disk blocks of the files. In contrast, scherlfs appends all files and updated meta data to the log. The same file layout is created only at another disk location. As the files are not deleted but only truncated, updates of directory information structures were not necessary. Thus, only the data which corresponds to the file updates is appended to the log. No directory data is saved in between the files by the appends to the log. As a consequence, in the reread phase the best achievable energy efficiency is reached because the requested data could be read from disk with the full transfer bandwidth of the hard drive. This explains the resulting energy savings.

The measurements show that scherlfs offers energy improvements especially for the creation of new files. Its energy efficiency is often better or at least comparable to that of other file systems in the read and write test phases.

B. Energy Consumption of Random Updates

The second comparative analysis examines the energy consumption of different file systems when small random I/O operations are issued to one large file. For this test, a file of 100 MB is created. This experiment consists of five phases:

- **Read**: All data is read in by issuing the appropriate amount of I/O operations with a size of 8 KB.
- **Rand read**: 24 MB of data is read in at pseudo random positions by issuing the appropriate number of I/O operations with a size of 8 KB.
- **Write**: 24 MB of data is updated at pseudo random positions by issuing the appropriate number of I/O operations with a size of 8 KB.
- **Reread**: The read phase above is executed again on the now updated file.
- **Rand reread**: The rand read phase above is executed again on the now updated file.

Figure 2 shows the energy consumption of the tests for the different file systems. As the target file is created on a new, empty file system, it could be organized in an optimized manner by each file system. Therefore, almost all

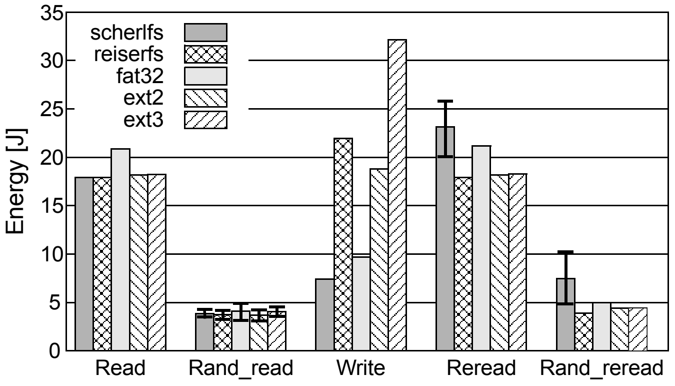

Figure 2: Energy Consumption of Random Updates

file systems show the same energy measurements during the first *sequential* and *random read* phases. However, fat32 consumes nearly 3J more than the other file systems in the read phase, due to the fact that a lookup in the file allocation table has to be performed every time a new block of the file is going to be loaded into memory. This requires two additional disk seeks if the corresponding FAT block has not already been loaded into the cache. Although the Linux kernel was modified to limit disk block caching it does not hinder file systems to explicitly hold disk blocks in memory. This is the fact for fat32 which caches a certain amount of FAT blocks. As a consequence, the additional seeks are only required when a FAT lookup affects a block which is not loaded into the cache at that time. As a consequence, the energy wastage is limited in the read phase and not as apparent in the random phase where uncached FAT lookups occur only with a certain probability. In the *write* phase scherlfs is able to perform with the lowest energy requirements because it only has to append the updated file data to the log. Fat32 is unable to perform with comparable results because the data has to be written to the updated locations which are randomly scattered within the file in contrast to being aligned in a sequential manner. This situation proves to be even worse in the case of the ext2 file system which organizes its data in different block groups. Therefore, its update-in-place method spends even more time seeking as it was the case for fat32. The two journaled file systems have to invest a significant amount of energy to perform their updates because of their integrated recovering techniques. Although scherlfs shows to be highly energy efficient for the initial read and write phases of this test its data layout becomes inefficient when random updates are performed. This can be seen in the *reread* and *rand_reread* phases. The energy consumption significantly increases after the file updates. This is due to the fact that the file data gets fragmented due to the append only behavior during the updates. File data is not aligned sequentially but is distributed within the log.

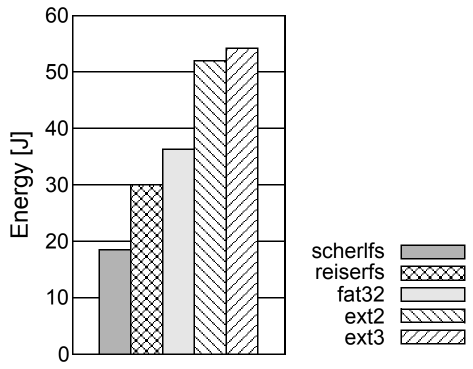

Figure 3: Digital Camera Appliance

C. *Testing a Digital Camera Appliance*

The next step was to examine the potential of our approach to save energy when running a real world application. Digital cameras record pictures and write them to disk immediately. Thus, this workload is characterized by mostly writing out mid sized files. With this test it was possible to compare the energy consumption of file systems when writing out new files. As digital cameras are embedded devices, we simulated their behavior on a PC. This is done by a test program that reads in 156 photo files (altogether 145.8 MB) from a digital camera with 3 megapixels. The average file size was 957 KB. Then all files are written out to disk subsequently.

Figure 3 shows the results of the energy measurements for the different file systems. They attest scherlfs an optimized energy use. In comparison to the energy consumed by reiserfs relative savings of 38% are possible with the new file system design. Scherlfs is able to write out the files with minimal overhead for updating meta data structures. Furthermore, nearly no disk seeks are required to create the files. As the created files are quite large the seeks which are required to update the directories are not visible in the measurements. Ext2 and ext3 show the highest energy consumption. This is caused by disk seeks which result from many meta data updates. For example, free block bitmaps, free inode bitmaps and the inode structures themselves have to be read in and updated quite often within the test. In conclusion, the experiment shows that significant energy savings can be reached when the overheads of traditional disk managers are avoided.

VI. Free Space Management

One important task for log-structured file systems is to manage the free space of the storage medium and make space occupied by old or deleted data inside the log available again.

There are two main approaches to free space management in a log-structured file system and both have similarities to common garbage collection techniques.

The first technique is to take valid data from the end of the log and overwrite unused space in the middle of the log. The log gets shorter and more compact and there is more space available at the end of the storage area. This is the *copying approach*.

The second technique is known as the *threading approach*. It fills all the available storage up to the end of the disk and then starts over from the beginning reusing the parts of the log which are no longer in use. This way, it creates a kind of interleaved log. This method involves less or no copying, but unused areas have to be identified and there may be fragmentation of frequently updated data.

A combination of both techniques is also possible, resulting in a *hybrid approach*.

As on-going work, we currently implement and evaluate different approaches to free space management which are discussed in the following sections.

A. *Copying Approach*

This method moves data from the end to the holes (obsoleted or deleted data) in the log. This way, the log is shortened and space is made available for future write operations. Since possibly large portions of data have to be moved around, this approach is not practical for incremental file system cleaning. A pitfall is that extensive copying may cause an inconsistent file system in case of a crash if not taken care of. To reduce the latencies caused by the cleaning process the copy operations could be performed incrementally.

First a source region has to be chosen from which the data will be moved into the log holes. Typically this will be the end of the log where new data is written to, but it may be beneficial in terms of lesser overhead due to copying to change this location. As a result the amount of valid data which will be moved is known. The second step is to identify a destination region in the log where enough space for the data is available, preferably an area with lots of holes in the log. If the copying step should also have a defragmentation effect, the full size of the files which are only partially in the destination area have to be taken into account. The same is true for files which are partially in the source region. The defragmentation step is optional and consists of reordering data in the destination area and copying data which belongs to the files in question to the appropriate places in the destination area. For this process, the free space in the destination area or the end of the log can be used as temporary storage. The final step is to move all data which belongs to the source region into the target area preferably keeping together data of individual files. In addition to that, the new end of the log has to be marked.

The following questions or parameters influence the decision of when and how the cleaner should be invoked:

- How much energy is available for cleaning?
- How much free space should be created?
- How much memory is available for temporary data during cleaning?

To sum up, the copying approach has the advantage that no overhead is incurred for writing since available memory at the end of the log is always contiguous. However, the cleaning process itself can come with a significant overhead both in energy and time. Therefore, the copying approach should be used favorably for on-line mode while no or at least few write operations are performed. If it is used during battery-powered operation, different policies for a copying cleaner are possible:

- Minimum number of copy operations.
  In this case, large areas of file data can be split to fill smaller holes in the log resulting in a higher fragmentation.
- Compacting cleaning.
  A defragmentation step is involved which joins matching extents of file data resulting in overhead due to copying.

B. Threading Approach

In the *threading approach* the log grows until it reaches the end of the disk and then starts over at the beginning, threading through the holes in the previous log while skipping all valid data. As a consequence, no explicit cleaning is necessary which makes this strategy well-suited for incremental use. The main drawback is that for each write operation a suitable hole has to be found. Thus the position and size of the unused space must be known during writing in contrast to the *copying approach* where this information is necessary only during the cleaning process. Furthermore, this method can cause significant fragmentation if there are only small holes left in the log. An explicit defragmentation step similar to the *copying approach* might become necessary. If there is still enough contiguous space available, writing complete files instead of only the updated parts could also prevent further fragmentation, especially if this would cause two or more small unused areas to be merged.

In our current implementation the holes in the log are managed using a pseudo file which considers the holes as its data. To reduce the size of this meta information we use extents (starting address and length of each hole). As a consequence, every update to a file's meta data also triggers an update to the meta data of the pseudo file, and few large holes in the log result in lower memory consumption for the meta information of the pseudo file than lots of small ones. Therefore, we try to minimize the fragmentation of the file data as well as the fragmentation of the holes by using a best-fit algorithm for finding suitable holes for new data. This involves sorting the holes by their sizes. Our current implementation is based on skip-lists for sorting.

C. Hybrid Approach

The main advantage of the *threading approach* is that no explicit cleaning is necessary. However, every write operation involves searching for a hole which is big enough to hold the new data. Furthermore, keeping such information in memory may be difficult. As proposed in [3] for LFS, a *hybrid approach* solves this problem by dividing the storage area into large chunks or segments. The *threading approach* is used at the granularity of these segments since it is easier to keep track of empty extents, using e.g. a bitmask. Eventually, partially used segments are cleaned up and merged using a *copying approach*, making free segments available for the log.

VII. Conclusion

We have presented guidelines for energy-efficient file system design. To prove our assumption we have implemented a novel, log-structured file system based on the idea of minimizing energy consumption by avoiding expensive disk seeks and latencies due to rotational delays. Energy measurements of several test scenarios prove the energy efficiency of the proposed data organization and layout. We have discussed several approaches to free space management and the implications of the design options on energy consumption. Recommendations for an energy-aware free space management for different energy settings are given.

As future work, we investigate the feasibility of an on-line adaptation of cleaning mechanisms or free space managers to changing access behavior and availability of energy. In addition to that, we experiment with low-power file system designs for other, possibly not block-oriented storage media.

VIII. Acknowledgements

The anonymous reviewers have helped us to improve this paper with their useful feedback.

References

[1] Czezatke, C., and Ertl, M. A. Linlogfs - a log-structured filesystem for linux. In *Freenix Track of Usenix Annual Technical Conference* (2000).

[2] Matthews, J. N., Roselli, D., Costello, A. M., Wang, R. Y., and Anderson, T. E. Improving the performance of log-structured file systems with adaptive methods. In *Proceedings of the sixteenth ACM symposium on Operating systems principles* (October 1997).

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

[4] Scherl, H. Design and implementation of an energy-aware file system. Department of Computer Sciences 4, student thesis SA-I4-2004-01, January 2004.

[5] Schindler, J., Griffin, J. L., Lumb, C. R., and Ganger, G. R. Track-aligned extents: Matching access patterns to disk drive characteristics. In *Proceedings of the First Conference on File and Storage Technologies FAST'02* (2002).

[6] Seltzer, M. I., Bostic, K., McKusick, M. K., and Staelin, C. An implementation of a log-structured file system for unix. In *USENIX Winter* (1993).

[7] Seltzer, M. I., Smith, K. A., Balakrishnan, H., Chang, J., McMains, S., and Padmanabhan, V. N. File system logging versus clustering: a performance comparison. In *USENIX Winter* (1995).

[8] Zheng, F., Garg, N., Sobti, S., Zhang, C., Joseph, R. E., Krishnamurty, A., and Wang, R. Y. Considering the energy consumption of mobile storage alternatives. In *Proceedings of the 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems MASCOTS 2003* (October 2003).