Reorganisation in energiebewussten Dateisystemen

Studienarbeit im Fach Informatik

vorgelegt von
Philipp Janda
geboren am 05. April 1978

Institut für Informatik,
Lehrstuhl für Verteilte Systeme und Betriebssysteme,
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:	Dipl.-Inf. Andreas Weißel
	Dr. Ing. Frank Bellosa
	Prof. Dr. Wolfgang Schröder-Preikschat

Beginn der Arbeit:	05. Juli 2004
Abgabedatum:	05. April 1978

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe, und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde.
Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 05. April 2005

Energy-Aware Reorganization in Log-Structured File Systems

Studienarbeit

by

Philipp Janda

born April 5th, 1978, in Heilbronn

Department of Computer Science,
Distributed Systems and Operating Systems,
University of Erlangen-Nürnberg

Advisors:	Dipl.-Inf. Andreas Weißel
	Dr. Ing. Frank Bellosa
	Prof. Dr. Wolfgang Schröder-Preikschat

Begin:	July 5th, 2004
Submission:	April 5th, 2005

Abstract

As mobile computing devices become more and more popular, the corresponding hardware and applications become more complex, comfortable, and thus energy hungry. Since the main advantage of mobile devices is their ability to work without a stationary power supply an important challenge has been to reduce power consumption, thereby prolonging the uptime of the devices in battery mode.

One way to reduce the energy needs of a mobile device is the adoption of a special data layout on its storage devices, which accounts for the mechanical mode of operation of a hard disk. Such a special data layout is, for example, a log-structured file system. The major disadvantage of a log-structured file system is that it wastes storage space during operation in order to reduce energy consumption.

To reclaim the wasted space, a particular free space management is necessary. While there are multiple possibilities with different advantages and disadvantages, all methods involve some overhead and an increased energy consumption. The question is whether the addition of such a free space manager will negate the energy efficiency of a log-structured file system and thus make log-structured file systems unsuitable for saving energy.

This work analyzes the properties of different free space management techniques and gives recommendations for their application in log-structured file systems. Implementations of variants of a certain class of free space management are described and tested for performance and energy consumption. Measurements of the cleaning processes were performed for different fragmentations and in comparison to other file systems. Although the two tested prototypes are similar, they show vast performance differences under certain circumstances.

Contents

1 Introduction **1**

2 Motivation **3**
2.1 Hard Drive Operation . . . 3
2.2 File Systems . . . 5

3 Related Work **7**
3.1 The Design and Implementation of a Log-Structured File System . . . 7
3.2 An Implementation of a Log-Structured File System for UNIX 8
3.3 Improving the Performance of Log-Structured File Systems with Adaptive Methods . . . 8
3.4 Considering the Energy Consumption of Mobile Storage Devices 9

4 Previous Work **10**
4.1 ScherlFS File System Layout . . . 10
4.2 Energy Characteristics of ScherlFS . . . 14

5 Free Space Management Strategies and Algorithms **15**
5.1 Copying/Compacting Cleaners . . . 15
5.2 Threading Approach . . . 18
5.3 Combined Approach . . . 19

6 Implementation **20**
6.1 The Copyclean File System Cleaner . . . 20
6.1.1 General Preparations . . . 21
6.1.2 Source and Destination Regions . . . 22
6.1.3 Compacting Cleaning . . . 24
6.1.4 Defragmenting Cleaning . . . 24
6.2 Future Work . . . 25
6.2.1 File System Support for Block-Inode Mapping . . . 25
6.2.2 Hole-Plugging Mode . . . 26
6.2.3 Threaded Approach in the ScherlFS File System . . . 27
6.2.4 Adaptive Cleaning . . . 28

7 Energy Measurements **29**
7.1 Preparations 29
7.2 Test Cases 30
7.3 Test Results 31

8 Conclusion **37**

Bibliography **39**

Chapter 1

Introduction

During the last few years, mobile computer devices and embedded systems have become more and more widespread. The main advantage of mobile computing for users is the greater flexibility of not being dependent on a fixed power supply. Due to the technological progress more useful and complex applications become possible, while the demand for such applications rises. Many of those applications require some sort of data storage. As the number of features of mobile devices such as cell phones, digital cameras or personal information managers with mobile office applications increases, so does the energy usage. Even though there have been various improvements regarding energy consumption and thus an increase in battery mode runtime, especially one area has so far been disregarded in connection with efficient energy usage: storage devices. One possible way to improve energy efficiency of storage devices without researching new materials for data storage or building new storage hardware is to change the layout and access patterns of data on common hard drive storage devices such as the Hitachi Microdrive[1].

In his thesis[2], Holger Scherl has examined and implemented an alternative means of organizing data on hard drives and thus minimizing energy consumption due to disk seeks and rotational latencies by using so-called log-structured file systems. While the prototype and the energy measurement results are very encouraging, the implementation still lacks a vital component for the ScherlFS which permits ongoing use of the file system. A log-structured file system saves energy at the cost of temporarily wasting storage space, thus an effective free space management is needed to reclaim the wasted free space areas.

This paper examines various ways of free space management in log-structured file systems as well as the effects of these techniques on the file system's performance and energy characteristics.

The following chapter (chapter 2) gives an overview of the current situation of storage media, file systems and their energy saving potential for embedded devices and explains the motivation for this thesis. Chapter 3

lists and summarizes papers which are related to the subject of this thesis whereas chapter 4 reviews the current implementation of a log-structured file system for the Linux operating system called ScherlFS [2], which is the basis for the implementations described in this thesis. In chapter 5 the basic strategies for free space management are analyzed and compared, while chapter 6 discusses an implementation of a certain kind of free space manager, a copying cleaner, for the ScherlFS file system as well as further details about other free space management techniques in relation to the ScherlFS file system. Chapter 7 outlines the test cases and measurements performed for the implementation of the copying cleaner and the ScherlFS file system in general and examines the results. Finally, in chapter 8 the test results and the information given in the previous chapters are summarized and interpreted.

Chapter 2

Motivation

As already indicated in the last chapter this thesis, and the previous work by Holger Scherl[2], which is described in detail in chapter 4 (page 10), try to reduce the energy consumption of storing and retrieving data on disk not by using some form of fancy new hardware, but by changing the organization and layout of the data on the device instead. This approach takes into account the way hard disk storage devices are constructed and work in order to reduce the number of hardware operations necessary to perform hard disk reads or writes.

2.1 Hard Drive Operation

While all modern hard drives are accessed via logical block addressing (LBA), this addressing method conceals the real design of hard disks. The older addressing method CHS (cylinder, head, sector) mirrors the actual hardware more closely, but was abandoned due to limits of the maximum size of hard disks.

Hard Drive Structure and Data Accesses A hard drive consists of one or more rotating magnetic disks and a read/write head which hovers above the rotating disks.

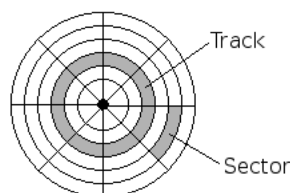

Figure 2.1: Hard drive geometry

The data on hard disk drives is organized in circles on the rotating disks. Such a circle is called a "track" and one such track can be accessed by the read/write head without moving the head to another position. Hard drives with more than one rotating disk usually have their disks stacked on top of each other and a read/write head for each disk. Often these read/write heads are not independent and can only move together. The set of the tracks on the rotating disks which can be accessed by all read/write heads without repositioning is called a "cylinder". Thus the combination of a read/write head and a given cylinder can identify a track in an unambiguous way. The tracks are further divided into sectors, which are the smallest data storage unit a hard drive can read or write (see figure 2.1). These sectors usually hold 512 bytes and are often called blocks (e.g. in logical block addressing). Therefore hard drives are commonly referred to as block devices.

If a block is going to be accessed on the hard disks the read/write heads usually have to be moved to the right cylinder—a process called seeking. If the heads are already in the right position this is a most favorable case since moving the heads is relatively time and energy consuming. Before the read/write heads can start accessing a disk block, typically one has to wait until the disk rotates into the right position. This delay is called rotational delay and is another important factor in runtime and energy consumption. Actual numbers on the influence of seeks and rotational delay on the performance and energy consumption of different hard drives can be found in [3].

As a consequence, data that is aligned on a single track can be accessed without intermediate seek operations and thus with reduced delay and energy usage. Sectors that are additionally located sequentially on the track minimize the rotational latency and thus further reduce access time and power consumption.

Energy Saving Modes of Hard Drives To support power management in mobile devices many modern hard drives feature low power modes, which can save energy while the disk is idle, but can result in severe performance degradation. A transition into or out of power mode usually takes some time and might cost additional energy as well.

A typical hard drive operates in one of four modes[4]: during the active mode the hard disk is ready to access data, but uses the most energy. If the hard disk is unused for a short period it changes to idle mode. In this mode it uses less energy but needs a short delay to switch back to active mode when a data request arrives. The standby mode is similar to the idle mode but requires still less energy but a considerable longer reactivation time. Therefore it is used for longer idle periods. Finally the sleep mode is used for long periods of system inactivity and uses little energy but typically requires several seconds before the drive can change into active mode again.

There are various algorithms responsible for choosing power saving modes (see [5] for an analysis), but in general it is most beneficial if the disk accesses are grouped together, thus maximizing the idle time periods and the periods the hard drive spends in low power modes.

2.2 File Systems

As far as hard drives are concerned, only raw data chunks of sector-size are read from or written to disk, but the user of computer systems usually further organizes his data into files and directories. Directories serve as containers for the files while files group related data and carry meta information with them.

A file system is part of the operating system and allows the user to create, remove, read, or modify directories, files and their meta information. It maps these constructs to raw data chunks which can be handled by the hard drive. A file system is also responsible for locating data on the hard drive and interpreting the data chunks as files or directories. This data organization causes common access patterns to the data, for example, usually the directory containing a file is read before the meta information and the actual data of the file itself. Furthermore, often the data blocks of a file are accessed together or in short time intervals.

Obviously, a file system that organizes its data in such a way that the data access patterns do not interfere with the raw data layout on the hard disk can offer better performance and energy consumption than other file systems.

A log-structured file system is an attempt to reduce hard disk seeks and therefore increase performance and energy efficiency.

Log-Structured File Systems Recently journaling file systems have become popular. These file systems maintain a special file called a journal or log to which all important updates are appended. In case of a crash the file system can be brought into a consistent state with the help of the log.

A log-structured file system uses the available storage space to record any updates similar to the log of journaling file systems. All update operations and, in case of the log-structured file system, all data is appended to the end of the log, causing it to grow until it fills all available space. Updates to existing data do not happen in place, but instead a new version of the data is appended. This has several advantages regarding the above mentioned energy saving options.

Since all data is written at the same position, a minimal number of seek operations is necessary for storing data. Naturally the stored data will most likely end up on the same hard disk cylinder. Due to this sequential data layout, future read operations will benefit as well. Data that is updated

successively and therefore might be related, will be stored in close proximity which in turn can improve later accesses. The reduced number of seek operations for read or write accesses will in general shorten the necessary hard drive activity time and thus enable longer idle phases utilizing low power modes.

The major drawback is that obsolete data in the log is not removed as it would require additional disk seeks and thus an increased energy consumption during updates. Therefore, it is possible that the log contains only a very small amount of live data while still filling up the whole disk space.

Free Space Management As a consequence a log-structured file system cannot be used over a long time period unless one has a way to reclaim the wasted space and thus shrinking the log. Depending on the type of free space management it could even be possible to restore the sequential data layout that might have been lost due to many small updates of file data.

This paper will examine exactly these possibilities.

Chapter 3

Related Work

Although log-structured file systems have so far been neglected in relation to energy efficiency, they have been noted for raw writing performance for many years. Since they also evenly distribute hard disk stress they are also important for some special sensitive storage media. As a consequence, log-structured file systems have been studied closely in the following papers for example. The last given paper is an exception as it deals with the energy consumption of mobile storage devices and file systems.

3.1 The Design and Implementation of a Log-Structured File System

In this paper[6] from 1992, Mendel Rosenblum and John K. Ousterhout describe their implementation of a log-structured file system called Sprite LFS and some simulations regarding the different policies of their free space management technique. For their prototype they chose to split the hard disk into fixed size segments and do a local defragmentation according to an analysis of the block usage in the segments. The file system skips full segments during writing. As a result the Sprite LFS achieves approximatley 70% hard disk throughput while conventional file systems only work at 5–10% efficiency. The remaining 30% are required for the free space management which was optimized using simulations of different management policies like the time when the cleaner should run, or how much space should be reclaimed on each cleaner run.

A crash recovery based on check points which denote a consistent file system state was added later on. As no data is overwritten in log-structured file systems, and due to the sequential file layout the crash recovery for the Sprite LFS is faster than in conventional non-journalling file systems.

As a result the paper presents log-structured file systems in general as being more efficient than standard unix file systems at this time.

3.2 An Implementation of a Log-Structured File System for UNIX

Three years later Margo Seltzer et al. redesigned the Sprite LFS to remove some deficiencies in the older prototype, e.g. the excessive memory usage and some shortcomings in the file system recovery and cleaning. Additionally, a better integration with the BSD tool set was desired. The new prototype, along with performance numbers on the cleaning overhead and comparisons to other file systems, are described in this paper[7].

The main changes to the old prototype are that the cleaner has been moved into userspace, and a modified segment layout on disk.

The bottom line of this project is that while the file system itself can utilize a large fraction of the available bandwidth, the cleaner can have a large performance impact. The new and old versions of the Sprite LFS work better with large amounts of main memory as buffers and asynchronous write operations which increase the contiguous chunks on the storage device.

3.3 Improving the Performance of Log-Structured File Systems with Adaptive Methods

As log-structured file systems offer excellent performance for most common workloads but suffer serious efficiency degradation in the event of heavy random updates and little idle time for cleaning, Jeanna Neefe Matthews et al. have tried to reduce these effects for the Sprite LFS and log-structured file systems in general. Their results are described in this paper[8].

The main idea is to make the file system more intelligent and aware of the current workload and the underlying hardware capabilities and to let it choose the best parameters for the current situation. Four concrete suggestions to this self-tuning approach are explained and measured using a file system simulator.

The first suggestion is specific to the Sprite LFS. It concerns itself with the proper choice of the segment size as too small segments may limit the transfer efficiency while too large segments can cause more cleaning overhead. The second suggestion deals with an alternative cleaning strategy called "hole-plugging" (see page 17 for further details) which is normally more inefficient than the normal Sprite LFS cleaner, but which is better suited for less idle times. Selecting the most appropriate one at runtime could increase the file system efficiency.

Normally the standard Sprite LFS cleaner reads one or more segments and writes the contained live data to an empty segment. The third suggestion extends this technique to segments that are already cached in memory. Naturally this also increases performance for large amounts of available memory.

Finally, a dynamic reorganization of the data layout to match read access patterns (as opposed to the write access patterns all log-structured file systems are well suited for) will improve the access for certain read patterns.

The given suggestions are especially effective for low-idle-time workloads where log-structured file systems normally perform badly.

3.4 Considering the Energy Consumption of Mobile Storage Devices

In this paper Fengzhou Zheng et al. analyze different file system design decisions in relation to the energy consumption during typical workloads. One of these files systems is a log-structured file system, while the other tested file systems are modified variants of a standard update-in-place file system. The modified versions provide different features that are normally attributed to log-structured file systems.

The authors use a logical disk system which enables them to test the different file system variants on different hardware. The tested hardware consists of a flash card, a microdrive, and a network card for remote storage. While the flash card has a very powerful energy management and the power management of the network card is independent of the workload, the microdrive mostly benefits from long periods of idle time, where the low power modes can save energy. Along with the log-structured file system, the variants with asynchronous writes and increased burstiness favor energy savings.

That includes the cleaning process, which should only clean free space on demand, or the idle periods will be disrupted. On the downside this will penalize the response time on heavily used file systems.

Chapter 4

Previous Work

This thesis is based on the work of Holger Scherl who implemented a log-structured file system called ScherlFS [2] for the Linux kernel using its virtual file system (vfs).

4.1 ScherlFS File System Layout

The virtual file system of the Linux kernel provides a framework and a common interface to storage devices for building different files systems. It uses two important data structures for handling devices and files: the superblock and the inode. The superblock contains all information about the file system itself and the device it is on, while an inode contains information about a file (e.g. access rights, modification time, the location of data blocks, etc.). Some file systems also use variants of these data structures to store this information on the device (most notably the ext2 file system) but this is not necessary as long as the required information can be gathered from the raw data on the device. The vfs provides hooks for reading superblock or inode information.

ScherlFS uses a superblock structure which is located at a fixed address (1024 bytes offset) on the device. All binary meta data is stored in little endian byte order by the ScherlFS file system driver. Figure 4.1 shows snippets of C code to illustrate the data that is stored in the superblock structure on disk. The most important elements of this structure are *s_blocks_count*, *s_log_block_size*, *s_magic*, *s_log_head* and *s_ifile_block*. The *s_blocks_count* field holds the total number of blocks on the device, while *s_log_block_size* defines how many bytes each block has (currently the value 1024 is hard coded in the file system driver). *s_magic*, the magic number of the file system, is used to identify a file system on a device and to ensure the right file system type during the mount operation (the ScherlFS magic number is 0xCA77). The field *s_log_head* denotes the position of the block which will be used in the next write operation. Every block with a smaller block number than

```
/* the scherlfs superblock on disk (located at byte offset 1024) */
struct superblock {
  uint32_t s_inodes_count;
  uint32_t s_blocks_count;      /* number of blocks on device */
  uint32_t s_free_blocks_count;
  uint32_t s_free_inodes_count;
  uint32_t s_log_block_size;    /* number of bytes per block */
  uint16_t s_magic;             /* scherlfs magic number: 0xCA77 */
  uint16_t s_mount_state;       /* some flags */
  uint32_t s_log_head;          /* next unused block at end of log */
  uint32_t s_log_tail;
  uint32_t s_ifile_block;       /* block number of .ifile inode */
};
```

Figure 4.1: The ScherlFS superblock structure on disk

s_log_head is used (or at least reserved) by the log, while all other blocks are unused. The field *s_ifile_block* is the disk block number of the inode of a special file called `.ifile`. This file is used by the file system driver to look up disk blocks for inode numbers, to everyone else it is of little use. The data in this file is treated as a large array of 32 bit block numbers (in little endian) and indexed by the inode number. The first four inode numbers (0-3) are reserved. Inode number two is the `.ifile` itself and the block number at this position is already obsolete—the correct block number is stored in the superblock structure. Inode number three is the inode of the root directory.

A directory is like an ordinary file but special directory entries are stored in its data blocks. Such an entry consists of the file name of the entry and the inode number for the file, along with some management information (the directory management is similar to the ext2 file system). The `.ifile` can be used to look up the inode block number for the inode number, and the inode itself can identify the data blocks for the file.

ScherlFS uses a structure shown in figure 4.2 to store important information about a file on the device. Most of the fields correspond to the fields of the in-memory inode structure used by the vfs except the *i_n_logchains* field. However, depending on the file type of the inode, which is encoded in the *i_mode* field of the inode, there is some more information following on disk right after this structure. For character or block device files the inode stores an additional 32 bit device number, for symbolic links the link target is stored as a character array, and for regular files and directories the inode holds an array of so-called extents. The length of this array is stored in the *i_n_logchains* field of the inode. Depending on the number of extents the complete inode data can span multiple (adjacent) disk blocks for regular files and directories.

```
/* a scherlfs inode structure on disk */
struct inode {
  uint16_t i_mode;        /* file type and permission bits */
  /* uint16_t _padding1; */
  uint32_t i_uid;         /* user id of file owner */
  uint32_t i_size;        /* file size in bytes */
  uint32_t i_ctime;       /* change time in seconds since 1.1.1970 */
  uint32_t i_mtime;       /* modification time */
  uint32_t i_atime;       /* last access time */
  uint32_t i_gid;         /* group id of owning group */
  uint16_t i_links_count; /* number of hard links to this file */
  /* uint16_t _padding2; */
  uint32_t i_blocks;      /* number of 512 byte blocks allocated */
  uint32_t i_flags;       /* some flags (currently unused) */
  uint32_t i_generation;
  uint32_t i_n_logchains; /* number of extents following */
};
```

Figure 4.2: A ScherlFS inode structure on disk

```
/* a scherlfs data extent on disk (aka. logchain) */
struct extent {
  uint32_t lc_block;  /* first block of the data chunk */
  uint16_t lc_length; /* number of blocks belonging to the chunk */
  /* uint16_t _padding; */
};
```

Figure 4.3: A ScherlFS extent for file/directory inodes on disk

An extent is used to identify a contiguous chunk of disk blocks containing file data. It consists of the first block number of the chunk and the length of the chunk in file system blocks. The logical block number of the file (the block number relative to the beginning of the file data), is stored implicitly and can be obtained by counting the blocks belonging to the previous extents in the inode. A C code definition of a ScherlFS extent can be found in figure 4.3.

As an example, the following steps need to be performed to read the file `/tmp/xxx.txt`:

Since the superblock structure is in memory when the file system is mounted, the block number of the `.ifile` inode can be looked up. The inode structure given in figure 4.2 has to be read and—since the file must be a regular file—the extents after that structure. As the `.ifile` inode is needed for almost every file access, the ScherlFS file system driver caches

it in the in-memory superblock structure for faster access. Since the next step is to read the root directory, the extents of the `.ifile` inode have to be scanned for the first file block because the root inode block number (inode number 3) is stored at the fourth array position in the `.ifile`, which is in the first file block. After that, the root directory inode and its extents can be read. Now all directory entries in the data blocks have to be scanned until one finds an entry with the name `tmp`. This entry, if it exists, also holds the inode number of the `tmp` directory. Again the `.ifile` inode and data blocks are used to look up the starting block number of the inode. After one has read the inode, ensured that the inode belongs to a directory and read the extents and corresponding file data blocks, a directory entry with the name `xxx.txt` needs to be found in the data blocks. If it exists, one can look up the inode block for the corresponding inode number, read the inode, read the extents and data blocks if the inode belongs to a regular file, and it is done. Most of this logic is handled by the virtual file system of the Linux kernel, along with some further management operations such as checking the file permissions. The ScherlFS file system driver only handles the parts that are specific to ScherlFS.

Updating a file is similar up to the point where one got the inode of the given file. On the first read or write access to a block of a file, the block is mapped to memory and put into a cache. If the block has changed since it was first mapped, it has to be saved back to disk, eventually. This is done by the virtual file system framework and the bdflush daemon (Linux kernel 2.4), so the ScherlFS file system driver only has to assign a new destination block number to the given block and mark the block as "dirty". The new block number will be the *s_log_head* value from the superblock, which will then be incremented. Assigning a new block number will also invalidate the extent the block belonged to, so this old extent needs to be removed, shortened or split. Generally a new extent for the block is needed unless the new block address is adjacent to the block's previous extent's data chunk, in which case the two can be merged. Since the extents and some other inode fields have changed, the inode structure has to be written to disk as well. This is also done using the vfs and a new block address. Then the `.ifile` data has to be updated to include the new inode address, in which case the corresponding `.ifile` block needs to be written to disk. This changes the `.ifile` inode which has to be written to disk, too, and that changes the address of the `.ifile` inode on disk. Thus the superblock field *s_ifile_block* has to be updated and the superblock saved to disk at its fixed position. In its current implementation the ScherlFS file system driver defers writing `.ifile` inode and superblock until the file system gets unmounted.

4.2 Energy Characteristics of ScherlFS

In the performance and energy measurements that were part of his thesis[2], Holger Scherl came to the following conclusions:

For sequential file operations such as creating new files or reading and writing large contiguous portions of a file, the log-structured approach of ScherlFS outperforms the reference file systems (fat32, ReiserFS, ext2 and ext3) concerning energy consumption for all tested file sizes (4 kb to 1024 kb). The reasons for these results are that all file and meta data can be written in a sequential manner, resulting in less disk seeks. Disk seeks are still necessary for the directory operations. Thus the above conclusion for small file sizes only holds if one uses a cache for directory entries, since for small files the directory lookups and updates affect the energy consumption significantly.

Regarding non-sequential or random access operations such as reading or updating many small regions scattered randomly all over the file, ScherlFS still gains some advantages due to the sequential layout of the file data on disk. However if there have been some updates to the test file the energy consumption rises. This is due to the fact that after various file updates the file data is no longer stored sequentially on disk. Although the performance is still comparable to the other file systems, the fragmentation causes serious performance degradation. Thus a means of restoring the sequential data layout of the file could improve the overall energy efficiency.

Chapter 5

Free Space Management Strategies and Algorithms

A log-structured file system uses a disk in an append-only manner and stores all data and almost all meta information at the end of the already used storage area. In case some data needs to be changed, an updated copy of the data is appended to the log along with the necessary updates of meta information, while the old (now obsolete) data remains at its original position and consumes storage space which could be used for valid data. This space is lost, since it is part of the log region and thus reserved by the log and there is no meta data on the file system that references this old data. If the log eventually consumes the whole disk space, the file system becomes unusable (or at least read-only), since no create, update or even remove operations are possible, because each operation requires appending data or meta data to the log. To keep the file system operational over time, one has to ensure that the wasted space caused by obsolete data in the log can get reused. This chapter introduces various strategies for free space management and discusses their advantages and disadvantages.

5.1 Copying/Compacting Cleaners

The most obvious solution to the free space management problem mentioned above is to move live data from the log head to the unused space in the log. This makes it possible to shorten the log and thus make room for new data at the log head. In comparison to the other free space management strategies, this technique does not enforce changes to the normal file system operations. There are multiple possibilities to choose the live data areas which should be copied and the unused destination areas where the live data should end up. The first step to move live data to unused space in the log is to identify the unused chunks. As mentioned before, no meta data references this obsolete data, so if the file system does not provide a special means of identifying

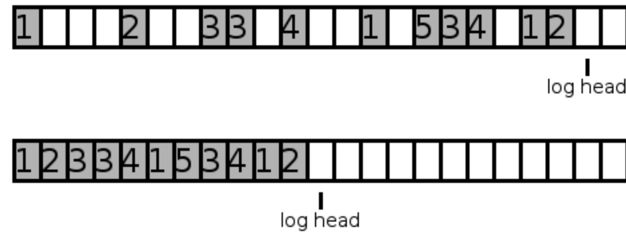

Figure 5.1: Compacting the log

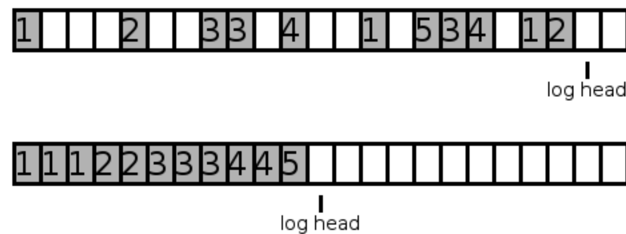

Figure 5.2: Defragmenting the log

unused space, one has to scan all live data and record all unused disk areas. Since the files that are (at least partially) located at the log head also need to be identified, all live data has to be scanned anyway unless the file system provides a reverse mapping from disk areas to the files that occupy them. After all important data and free space areas are identified, one can start moving data around.

Compacting The easiest way is to just overwrite the unused space with the live data that is next to the unused region, starting from the log beginning. All unused space will accumulate at the log head which can then be set to the end of the live data. This process is outlined in figure 5.1.

Defragmenting Since a log-structured file system benefits significantly from the sequential data layout of its files, an obvious enhancement to the proposed compacting of live data is to gather all parts of a file and move complete files only. This could eliminate fragmentation at the cost of even more copying and an increased complexity of the cleaning algorithm: This variant could involve moving file parts that are not at the log head and one has to ensure that the unused areas are large enough to hold the complete files. This cleaning strategy is demonstrated in figure 5.2.

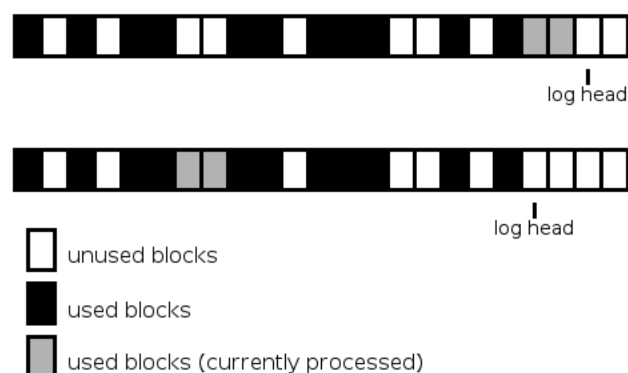

Figure 5.3: Hole-plugging

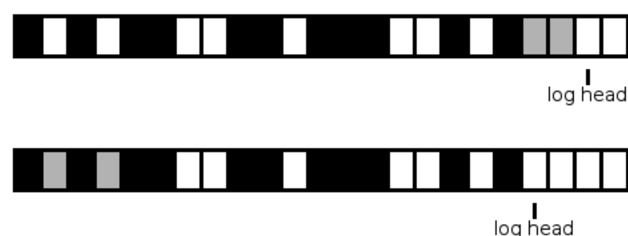

Figure 5.4: Hole-plugging with data chunk splitting

Hole-Plugging There is a technique for copying cleaning which reduces the restructuring of the live data in the log and therefore minimizes the copying overhead: "hole-plugging". This algorithm determines the size of a live data area at the log head and searches for a region of unused space that is large enough to hold that data chunk. Then the data is moved and the log head reset (see figure 5.3). If there is not enough contiguous free space in the log, there are several possibilities to move the live data: One could split the live data and scatter it through available unused areas (see figure 5.4), but this would result in an increased fragmentation of said data and thus a decrease in performance. Another possibility is to use a local compaction as in the first mentioned variant of the copying cleaner to accumulate enough free space somewhere in the log for the whole data region. While this involves more copying it also limits fragmentation and can even be modified to use a local defragmentation step instead of a simple compaction.

As this algorithm features the smallest copying overhead and, in its pure form, only processes one live data area and accordingly only one file at a

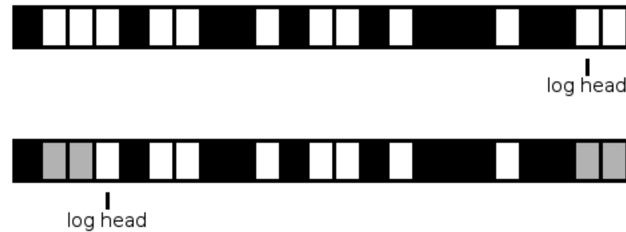

Figure 5.5: Threading approach - four new data blocks are appended to a nearly full log

time, it is the most suitable alternative of the previously mentioned strategies for incremental use and therefore capable of cleaning the file system while it is in use.

While the copying solution is conceptually the easiest free space management strategy, it also involves much copying and thus consumes more energy than the other techniques. Since it touches many live data regions, it is generally difficult to provide correct synchronization for incremental use while the file system is active. Its advantages are that it minimizes defragmentation of the log, and maybe even its files, while imposing no restrictions on the normal file system operations. There is no additional overhead for writing to the log since there is always contiguous memory at the head of the log. The copying solution also has some similarities with copying garbage collection algorithms (see e.g. [9]).

5.2 Threading Approach

In this approach the log grows normally until it first reaches the end of available disk space. Then it starts over from the beginning of the log using only the unused log areas and skipping all live data sections. Therefore, it creates an interleaved log (see figure 5.5 for an illustration). 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 area of unused space has to be found in the log, and the position and size of all unused areas in the log need to be known during writing. In contrast, in the copying approach this information is necessary during cleaning only.

Once the unused log areas are known and the current live data chunk is too large for the next free space area at the log head, the same considerations as in the "hole-plugging" variant of the copying approach apply: One can split the live data or do a local compaction or defragmentation of the live

data around the destination area. A third possibility is to skip all undersized free space regions, use the next free space region that is big enough and try to reuse the skipped areas when the log restarts from the beginning next time.

As the threading method can cause significant fragmentation especially of frequently used files if there are only small holes in the log, an explicit defragmentation step similar to the copying approach might become necessary. Furthermore, this approach could involve a large memory overhead for many non-contiguous free space regions in a large log in order to keep the free space information in memory. Given that one has to choose suitable free space areas for a given chunk of live data this approach is similar to common memory allocation strategies (see e.g. [10]). The chosen algorithm determines which free space area to use for new live data and which data structures to use to organize the list of unused log areas.

5.3 Combined Approach

As the threaded approach is well suited for incremental use and is also very energy efficient, while the copying approach incurs significantly less fragmentation and a more sequential data layout, it is tempting to combine the two methods to get the best of both worlds. In section 5.2 about the threading approach, some local enhancements to the threading approach utilizing compacting or defragmenting methods are already mentioned. There is still another severe problem with pure threading: the large memory overhead due to the list of unused log areas, which can become quite high for large disks and many small free space areas. One way to solve this dilemma is to partition the available disk space into larger chunks and use the threading approach for these areas. As there are now less areas the in-memory list and therefore the memory overhead becomes more acceptable. For the data in the chunks a copying approach is taken along with some options of merging two sparsely populated chunks. See [6] and also section 3.1 (page 7) for an implementation and detailed discussion of this technique.

Chapter 6

Implementation

This chapter describes the implementation of a copying cleaner for the ScherlFS file system. For an introduction to the copying cleaners see section 5.1 (page 15). For details about the ScherlFS file system implementation refer to chapter 4 (page 10).

The cleaner supports two operation modes: a defragmenting mode and a simple compacting mode. As the file system and thus the cleaner are intended for embedded or mobile systems, energy and main memory usage have been taken into account. Since mobile devices could run out of power or be switched off by the user extra care has been taken to ensure a consistent file system and no possible data corruption in case of a power loss or crash. The algorithms, data structures and techniques for the implementation are described in the following sections.

6.1 The Copyclean File System Cleaner

The presented copying cleaner is a user space program as opposed to a kernel module for the following reasons which have already been indicated in chapter 5: as copying cleaners need exclusive access to almost every file on the device, it is difficult to maintain normal file system operation anyway. A single update operation during cleaning would annihilate the whole effect of the cleaning run, as updates cause new data at the log head which effectively prevents shrinking the log. The various caching mechanisms in the Linux kernel further complicate copying file data. As the compacting cleaner already needs lots of energy due to extensive copying, it is best used when recharging the mobile device. The defragmenting mode needs even more energy. Further advantages are the improved portability as the cleaner is not tied to a specific kernel version or even a specific operating system (a POSIX operating system is required for the current prototype), and superior stability of the kernel since possible bugs in the cleaner cannot effect kernel operations.

The cleaner program gets the required number of blocks it should make available for future updates as a command line argument and tries to shrink the log by the given amount. It is imaginable that the cleaner could decide on its own how much cleaning is necessary, but the current implementation does not support this.

6.1.1 General Preparations

The two operation modes of the cleaner program share some similarities, especially at the beginning of the cleaning process. Both modes try to shrink the log at the log head to enable further updates to the log data, and thus need to define a region at the log head which will be freed from live data—the source region. The live data from the source region will be moved to somewhere in the log where enough unused space is available—the destination region. The current cleaner prototype places the destination region at the very beginning of the log since the data in this area is the oldest data in the log and the area will most likely contain much obsolete data and thus much reusable free space.

While choosing source and destination region is slightly different depending on the cleaning mode (see relevant sections below) there are still common prerequisites.

Unless there is some kind of kernel support for a reverse mapping of disk blocks to the corresponding file, the cleaner has to scan all existing inodes present on the device to create such a mapping on its own. The current ScherlFS implementation does not provide such support but see section 6.2 for further thoughts about this matter. As such a mapping can be quite large for big devices (for every disk block of 1024 bytes a four byte inode number is required [1]) and main memory might be limited on embedded systems, the cleaner prototype can limit the created mapping to the two areas where the source and destination regions will most likely be. Especially if the requested free space is small, this could save significant amounts of memory, however all inodes still need to be scanned so no time or energy savings can be achieved in this way. But as a consequence, the possible size of source and destination region is limited by the size of the available mapping, which could result in less than the required reclaimed free space.

Since the disk block addresses of all inodes are stored in the `.ifile`, scanning the inodes first involves reading this file. The disk block number of the `.ifile` inode is stored in the superblock structure which in turn is

[1] For a 2 gigabyte large device, which is the biggest supported size of the current ScherlFS implementation, a complete block inode mapping would consume

$$(\frac{2\,gigabytes}{1024\,bytes}) \cdot 4\,bytes = 8\,megabytes$$

of main memory.

located at a fixed position on the device. The `.ifile` data, which holds the disk block addresses, can be accessed through the extents of the inode.

Once all inodes have been located, the cleaner builds the block-inode mapping. The program starts with a mapping where all disk blocks are considered free. The mapping is simply an array of n inode numbers, where n is the number of disk blocks on the device. Then the cleaner reads the inodes one after another and marks the disk blocks which belong to the inode in the mapping array. The starting address is the disk block number in the `.ifile` data; the block size of the inode can be calculated, depending on the inode type, using the information in the common inode header (see chapter 4 (page 10) for details). If the current inode is a regular file or a directory, the data blocks given in the inode extents are marked too. After all inodes have been scanned, the elements of the mapping array either hold a valid inode number (the inode the disk block belongs to), or are empty.

The block-inode mapping is later used to determine the source and destination region for the cleaning process.

The current prototype reads the `.ifile` data into memory for faster access to the inodes, but if there are many files on the device the `.ifile` data could become very large (4 bytes for every file on the disk). To reduce memory usage it would be possible to read only parts of the `.ifile` (e.g. one disk block) at a time.

6.1.2 Source and Destination Regions

Once the disk block-inode mapping is complete the program can start to determine the source and destination region using the information in the mapping array. Such an area is just a range of disk blocks. Since the cleaning process should shrink the log, the source region has to start at the log head, consisting of zero blocks. It will grow toward the beginning of the log in order to fulfill the free block request. The destination region starts at the beginning of the log and grows in the direction of the log head. The current prototypes handles source and destination region alternately depending on which region needs to grow. Should a region meet the end of the mapping (especially in case of the limited mappings due to little main memory) or reach the other region, the process stops even if not the whole requested amount of free blocks can be fulfilled. Otherwise the process stops as soon as the source region is as large as the requested number of free blocks and the destination region has enough free space for all live data that has to be copied.

The cleaner will consider whole data chunks, i.e. groups of disk blocks, that are either empty or belong to the same inode. If the current data chunk is empty and belongs to the source region, the region can grow by this amount. If it belongs to the destination region, the region can grow but also the space for live data that can be copied from the source region

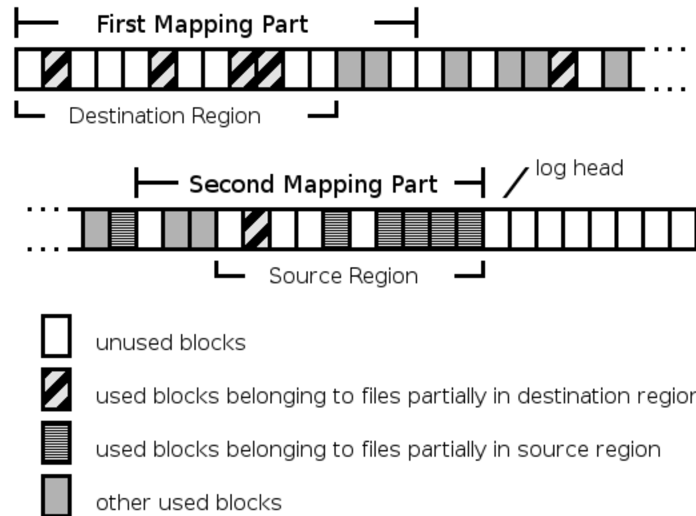

Figure 6.1: Example for source and destination regions in defragmenting mode

increases by the number of blocks in the empty chunk. If the chunk is not empty, then the following steps, depending on the cleaning mode and source or destination region, are necessary:

Regions in the Compacting Cleaner For the compacting cleaner, a non-empty chunk in the source region causes the region to grow normally, but the number of free blocks required in the destination region increases by the size of the chunk. A non-empty chunk in the destination region simply results in the growth of the destination region.

Regions in the Defragmenting Cleaner The case is slightly more difficult for the defragmenting mode. Since the whole file will be copied to the destination region in the course of defragmentation, one has to take the whole file size and the inode size into account. The current implementation of the cleaner needs to reread the inode from disk to gather this information, but if there is enough main memory available it is possible to cache these values during the inode scanning for the block-inode mapping. If the non-empty chunk is in the source region, the region grows by the size of the chunk but the number of required free blocks in the destination region is increased by the whole number of blocks the file occupies on disk. If the chunk is in the destination region, the region grows by the size of the chunk and the number of available free blocks in the destination area is decreased

by the number of blocks of the rest of this file. As the whole file size is counted for a non-empty data chunk, one has to make sure that the file does not get counted more than once even if there are multiple chunks on the device belonging to this file. Therefore, a bit set for each region is maintained where already handled inodes are marked. Data chunks of marked inodes are considered as being empty chunks because the corresponding file has already been handled. If parts of a file are both in the source and in the destination region it is important that the file is only marked in the bit set of the destination region (see the defragmenting algorithm below for reasons). Figure 6.1 shows an example.

Since until now no file system modifications have occurred the data is still in a valid state. The following operations will have to take special care to keep the data consistent in the event of a sudden power loss.

6.1.3 Compacting Cleaning

The block-inode mapping and the source and destination regions include all necessary information to start the compacting cleaning process. The program searches for the first non-empty chunk of data at the beginning of the destination region. Additionally, it counts the number of free blocks before this chunk. In case the chunk is already at the beginning of the destination area, the beginning of the area can be repositioned to the end of the data chunk. Else if the free space at the beginning of the destination area is big enough, the data chunk can be moved to its new position, which will cause in-place inode updates for the corresponding inode as an inode data extent will be changed in the process. If the free space is not big enough, the file chunk needs to be copied to a temporal location at the end of the log. This involves in-place inode updates and a superblock update since the log head changes. Theoretically, the chunk could be moved back immediately afterwards, but delaying this might prevent the need to temporarily copy the next data chunk. Therefore the chunk is just added to the source region and processed later. In every case the block-inode mapping has to be updated.

These steps are repeated until no more live data is present in the remaining destination area. Then the live data of the source region can be copied chunk-wise to the empty destination area, and the log head can be set to the beginning of the source area which will shrink the log.

6.1.4 Defragmenting Cleaning

The algorithm for defragmentation is similar to the compacting method described above. As above, first the destination area is processed from the beginning. For the first live data chunk, the cleaner checks whether the free space before the chunk is big enough to hold the complete file. If this is the

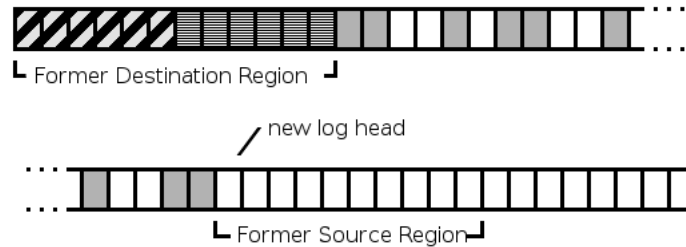

Figure 6.2: Regions shown in figure 6.1 after defragmentation

case, the file is copied and the relevant meta information is updated. This also includes the block-inode mapping and the `.ifile` data. The beginning of the destination region is moved by the number of blocks allocated for the corresponding file.

If there is not enough space and the file is not already completely at the beginning of the destination (e.g. due to prior defragmentation operations), the complete file or, if the file is too big, at least the data chunk for the relevant extent has to be moved to a temporary storage region at the log head. This results in inode updates, `.ifile` updates, block-inode mapping changes, and a temporary growth of the log and therefore superblock updates. The moved file is marked in the bit set of the source region for later processing.

If all live data chunks in the destination area have been processed and thus the remaining destination region is empty, all files marked in the bit set for the source region are copied completely to the destination region. Afterwards, the log head is reset and the relevant meta information is updated.

Figure 6.2 shows how figure 6.1 would look like after the defragmenting step.

6.2 Future Work

6.2.1 File System Support for Block-Inode Mapping

Since the block-inode mapping mentioned above is necessary for both cleaning modes (and in general for all copying cleaners), and creating the mapping is rather expensive because it involves scanning all inodes on disk, it is beneficial to provide this mapping as part of the kernel file system driver. The mapping could be generated during file system creation and continuously updated by the kernel file system driver as part of write operations. To ensure that the mapping data outlasts computer reboots and furthermore is available in an unmounted state to cleaner programs a special file similar

to the `.ifile` is imaginable (a proper name would be `.mfile`). The data of the `.mfile` would simply be the mentioned array of inode numbers where the index into the array would define the block number on the device. To prevent serious performance degradation, the `.mfile` would not be saved on every file system update, but instead periodically after longer idle periods and especially before file system unmounts to provide a consistent mapping to the cleaner programs. Even if there are further updates (e.g. `.ifile` updates) to the file system after the `.mfile` has been written, or in case of a file system crash before the updated `.mfile` can be stored to disk, it is relatively easy to bring the mapping data to an updated state given only the old `.mfile` data and the `.ifile`. One simply has to recalculate the occupied space for all files whose inodes are stored after the `.mfile` inode in the log. As new data is always appended to the log, the `.mfile` inode marks the point in time where the mapping was still accurate. All newer data comes after that point and therefore has to be reprocessed. The inode positions can be determined from the `.ifile`.

In summation, the kernel file system driver could provide an up-to-date block-inode mapping at the cost of slightly increased management overhead during write operations and rare additional data updates. But even an outdated mapping could speed up the inode scanning phase for both copying modes considerably, especially if there are many files on the disk. The extra required storage space for the mapping would be 1/256 of the storage capacity of the disk.

6.2.2 Hole-Plugging Mode

The cleaner program could be modified to use the hole-plugging approach as described on page 17. The first steps of this technique are similar to the compacting cleaner described above: The block-inode mapping is needed and the source and destination region can be determined exactly as in the compacting case. As it is important to identify the best matching free space area in the destination region to each live data chunk in the source region, a sorted data structure for free space areas in the destination region and one for the data chunks in the source region is necessary. The best course of action is probably to first serve the perfect matches from largest to smallest chunk. Later one could try to combine the remaining live data chunks to fill one of the remaining free space areas. Eventually, one will end up with small free space areas and bigger data chunks.

Since the compacting of live data in the destination region (and thus the merging of the free space areas) could involve significant copying overhead for a small number of remaining files, the preferable solution is probably chunk splitting. As long as this does not cause an inode to grow due to an increased number of extents, this method is rather straightforward. If there really is not enough contiguous space for the bigger inode it has to remain

in the source area.

The energy efficiency of this method in comparison to the other copying solutions would be interesting, as the hole-plugging strategy has the least copying overhead.

6.2.3 Threaded Approach in the ScherlFS File System

While this thesis generally concentrates on copying cleaners the threaded approach for free space management mentioned in section 5.2 (page 18) can be added to the ScherlFS implementation without much difficulties. As described in this section, the file system driver needs to know the free regions of the log during all write operations. This data has to be maintained and updated, thereby causing minor management overhead. Since the free space information has to outlive system reboots, one can benefit from the existing extent management of the ScherlFS inode implementation. A special file (called e.g. `.ffile` following the `.ifile` naming convention) can be used to track free space areas and store them in the file extents. As it is considerably more difficult to update an old `.ffile` given only the changed files and the `.ifile` than for the `.mfile` case above, the `.ffile` inode must be written to disk on every file data update or the file system consistency in case of a crash will be sacrificed. However if the above proposed `.mfile` is available, the updates of the `.ffile` could be delayed until the file system unmount. If there is ever a power loss or crash, the `.mfile` can be updated as described above, and the `.ffile` can be rebuilt using the block-inode mapping. An easier implementation probably would not save the `.ffile` at all, but instead rebuild it on every mount using the `.mfile` at the cost of a one-time increased read overhead. Using these two special files together could also enable the file system driver to switch threading on or off on demand. This would be especially useful for adaptive cleaning (see below). Since an inode holding many extents will occupy more than one disk block this write overhead could be considerable especially for many small updates (and thus the resulting small free space areas).

Finding an appropriate free space area in case the log already wrapped around the device could also cause performance loss for write operations, but a more efficient data structure which could speed up the free space lookup for a given chunk size could prevent that. For example, collections ordered by size like trees or a skip list could have a positive effect on finding suitable free space areas. Also note the similarities to memory management in general as described in e.g. [10] and the resulting possible data structures and algorithms. On the other hand, the file data can always be split into single blocks—only the inodes might need continuous free space areas larger than one disk block. Of course, this would destroy one main benefit of log-structured file systems: the sequential data layout.

6.2.4 Adaptive Cleaning

As the mentioned free space management algorithms all have different advantages and disadvantages, it could be beneficial to let the file system choose a cleaning method during runtime according to the current situation. This might involve choosing hole-plugging, compacting or defragmenting mode depending on the available energy and the fragmentation of the device.

If an existing threading mode can be deactivated on demand (using the block-inode mapping to rebuild the free space information at the next reactivation time, as described above), this could also save energy in case the file system has just been compacted or defragmented.

Another area where the file system driver could enhance cleaning efficiency is choosing the destination region for compacting or defragmenting mode. A large region with much live data at the beginning of the log could be skipped since compacting or defragmenting this area would result in many copy operations for very little gain in terms of reclaimed free space. The block-inode mapping of the `.mfile` would provide the required information for this optimization step. If there is no `.mfile`, the storage space could be divided into a fixed number of segments and usage information for these segments could be stored, for example, in the superblock structure. A copying cleaner could choose the most sparsely populated segment as destination region.

Chapter 7

Energy Measurements

This chapter describes the measurements performed on different file systems as well as for different cleaning strategies in ScherlFS file systems.

7.1 Preparations

The energy and timing measurements were performed on a Hitachi Microdrive. A four-channel analog-digital converter measures the voltage drop at a sense resistor in the 5 V line leading from the power supply to the hard disk and transfers this data via the parallel port to another computer where the results are recorded. The voltage drop is acquired with a resolution of 256 steps and at a rate of up to 5000 samples per second.

To measure the energy characteristics of the file system without disturbing influences like buffers or hardware low power modes, the linux page cache and the APM mode of the microdrive are disabled/limited for the duration of the tests using a kernel patch implemented by Holger Scherl for his thesis[2] and the hdparm utility, respectively. Additionally, before every file system test, the partition is unmounted to flush existing caches and remounted with clean buffers.

Fat32, ext2, ext3 and ReiserFS serve as reference file systems for ScherlFS.

To simulate the effects of file system usage in a predictable and reproducable manner a perl script has been implemented that issues a variety of file operations on a mounted file system. The fragmentation tool can create and remove a specified total amount of new files with random file size and contents (within a specified range) and do in-place modifications of random size in existing files. A recording mechanism allows one to replay the same fragmentation for different file systems and different settings.

For the evaluation of the different file systems and the relative costs of cleaning, two special fragmentations are used that are composed of small, medium, and large files and random updates on the data. For the first fragmentation, subsequently called ”standard fragmentation”, a total amount

Fragmentation Type	Files	Live Blocks	Dead Blocks
Standard Fragmentation	1128	110546	3354
Coarse Fragmentation	93	95656	271
Cleaner Fragmentation (small files)	2085	12182	7038
Cleaner Fragmentation (medium files)	37	8610	116

Figure 7.1: Characteristics of the different fragmentation samples

of 5Mb of small files (1 kb–10 kb), 20 Mb of medium files (50 kb–500 kb) and 75 Mb of large files (1 Mb–10 Mb) are created. Afterwards a total amount of 1 Mb of medium sized updates (10 kb–50 kb) and 256 kb of small sized updates (1 kb–5 kb) are performed on random files at random locations. For the write tests, the above steps are measured individually. The second fragmentation uses the same values but excludes small files and updates to limit the fragmentation somewhat, and is hence referred to as "coarse fragmentation".

As it has turned out during the measurements, the cleaners (especially the compacting cleaner) show horrible performance with deactivated page cache: the cleaning of the chosen fragmentation is too time consuming. Thus, a reduced fragmentation is used to compare the two cleaning modes. To measure the impact of file/chunk sizes on the cleaning process, one fragmentation with approx 10 Mb of small files (1 kb–10 kb) is created, and immediately afterwards 20% of the file data is removed again. The remaing files are modified randomly using the in-place mode of the fragmenting utility with a total amount of 5 Mb and random updates between 1 kb and 5 kb. As the block size of ScherlFS is 1 kb, smaller files do not make sense for this measurement. A similar fragmentation is created using the same total amount of file data and updates, but for medium sized files (50 kb–500 kb) and modifications (10 kb–50 kb). This results in less files on the device.

The different fragmentation characteristics (in terms of the ScherlFS file system) are listed in figure 7.1.

7.2 Test Cases

As this thesis tries to estimate the impact of different free space management techniques on the energy consumption and performance of the log-structured file-system ScherlFS, first the reference file systems are compared to ScherlFS without any free space management (see [2]). First the file creation and update performance for artificial test cases are measured (figures 7.2 and 7.3). The details are described in the previous section, as the parts of this test correspond to the single steps of the "standard fragmentation" mentioned above. This can show how much energy ScherlFS can spend for free space management without being worse at energy consumption than

the reference file systems.

As the chosen fragmentation and, as a consequence, the cleaning mode can influence the reading of files on all file systems, especially on a log-structured file system, the read performance for the different file systems and the two fragmentation alternatives are measured (figure 7.4) by simply reading all files that were previously created and modified using the first test. This may point out future energy advantages when performing a specific cleaning mode. In this test case a modified version of ScherlFS that uses a simulated directory cache is listed. See [2] for a rationale on this matter.

The next test measures the performance and energy usage of the two implemented cleaning modes (figure 7.5) for the two special fragmentation situations mentioned in the previous section. After that, the subsequent read behavior for ScherlFS is measured for the cleaned log. To enable direct comparisons to the previous read tests, this test uses the same fragmentation as above (figures 7.6 and 7.7). The results for the compacted log were obtained by temporarily enabling the Linux page cache during the cleaning process.

As the previous cleaning tests already cleaned the whole log, the following tests measures the performance and energy consumption when reclaiming only 10% of the available free space. This test also uses the normal fragmentation. The compacting mode performs reasonably well for this limited cleaning so its numbers are included.

Finally, in the last test, the possible gain by implementing a kernel based block-inode mapping as explained in section 6.2 is examined.

7.3 Test Results

The first two figures (7.2 and 7.3) show the energy and time consumption of the different file systems during the test. The results are similar to those of Holger Scherl in his thesis[2] and show that log-structured file systems in general perform well on write operations, due to the reduced number of disk seeks. The figure shows a total energy saving of approximately 105 J in relation to fat32 which is the second best file system in this test.

The bad performance of the ReiserFS file system is surprising. Previous measurements have shown this file system as being comparable to ext2 or ext3. The reasons for this are yet unknown but it may be an effect of a new ReiserFS version or the hardware. The tests previously performed by Holger Scherl applied to the given setting show similar results.

The fact that ScherlFS performs worst in the fragmented read test (figure 7.4) is not unexpected. The file operations cause the files on a ScherlFS device to be scattered in small chunks all over the log. The energy advantage of sequential data layout and a reduced number of disk seeks is negated. As

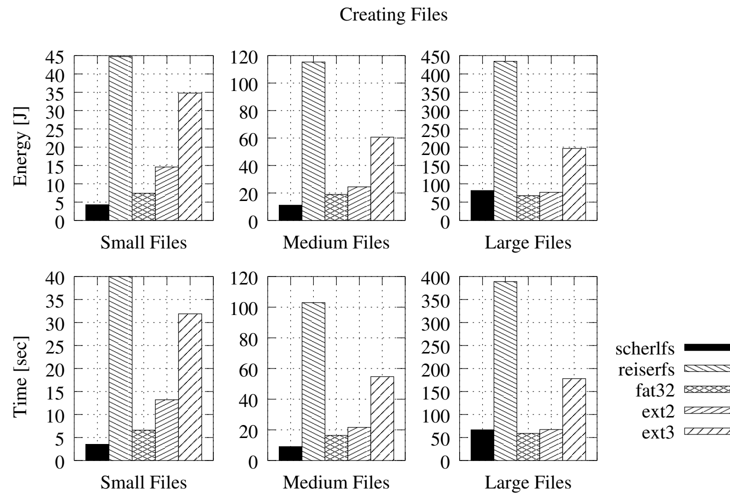

Figure 7.2: Energy consumption and time needed for creating new files

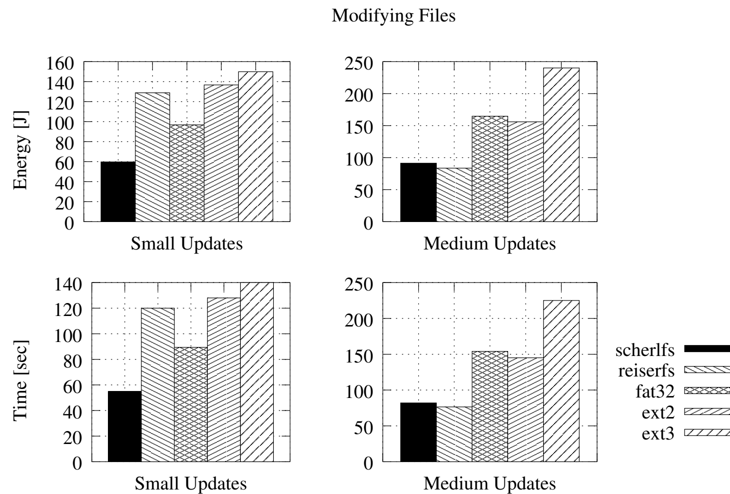

Figure 7.3: Energy consumption and time needed for updating files in-place

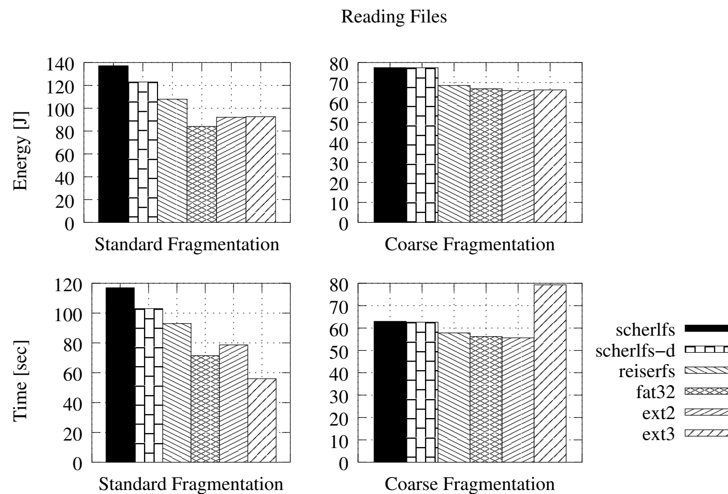

Figure 7.4: Performance characteristics for reading

the figures 7.6 and 7.7 show, an improvement in read can be reached. This could also mitigate the energy cost of cleaning.

The energy cost of the two cleaning modes for different fragmentations and different buffer sizes are shown in figure 7.5. The buffer size determines how much data can be read into main memory and therefore how many seek operations are necessary to move a data chunk or file to a new position. As can be seen in the figure the compacting cleaner is relatively independent of the size of the files since it is more concerned with contiguous file chunks. The copying cleaner in contrast reads and buffers a single whole file, if possible, and exhibits degrading performance as the number of files increases. A further optimization for the defragmenting cleaner prototype is to collect more than one file in the internal buffer in case it is large enough. In theory, the small buffer size of 2 kb should seriously affect the defragmenting cleaner, as most files cannot be buffered anymore. Since the compacting cleaner currently only collects contiguous data chunks belonging to file extents which are rarely larger than two blocks anyway, a smaller block size should not hinder its operation significantly. Surprisingly, even for larger files the defragmenting cleaner prototype does not suffer a noticeable performance degradation.

In figures 7.6 and 7.7 the influence of file data fragmentation on ScherlFS file systems is examined. It is shown that compacting the log can decrease the necessary time and energy consumption by about 40% in the "standard fragmentation" case. In this test case defragmenting the log, which should normally further improve the sequential data layout, had no positive effect compared to compacting the log.

For compacted or defragmented logs, the read performance and energy

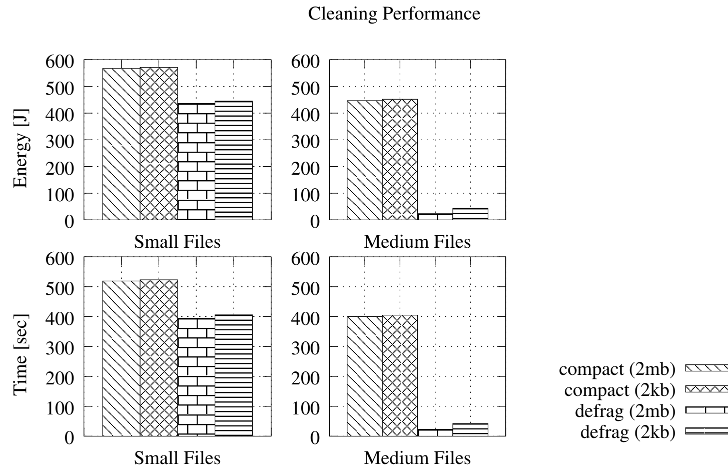

Figure 7.5: Performance of the two cleaning modes

consumption of ScherlFS is again better than for the reference file systems shown in figure 7.4.

Until now all cleaning processed the whole log. This has shown to be disadvantageous for the compacting cleaner, which requires more disk seeks due to its lack of a buffer mechanism. The defragmenting cleaner buffers whole files in memory before writing them back to disk, thus it can save two disk seeks for every extent compared to the compacting cleaner. A natural optimization of the current compacting cleaner implementation would therefore be the addition of data buffering.

However figure 7.8 shows the energy consumption and performance of the two cleaner prototypes applied to the "standard fragmentation". As already mentioned, the compacting cleaner performed so poorly with disabled caches that the test has been aborted.

In the case where only 10% of the wasted free space should be reclaimed, the compacting cleaner catches up with the defragmenting prototype. The performance is still not comparable, although the defragmenting cleaner obviously touches almost all files on the file system (the energy consumption and time is equivalent to the complete cleaning of the log). As a side note, if the caches are active however, the compacting cleaner outperforms the defragmenting cleaner in the second case.

Finally, in figure 7.9 the cost of the block to inode mapping necessary for both cleaning modes is put in relation to the total cleaning. It shows that support for such a mapping in the file system could improve the cleaning process. In this case the performance gain would be moderate, but with an increasing number of files on the device such an `.mfile` could be favorable. The fragmentation used in this figure is the "Cleaner Fragmentation (small files)" in table 7.1.

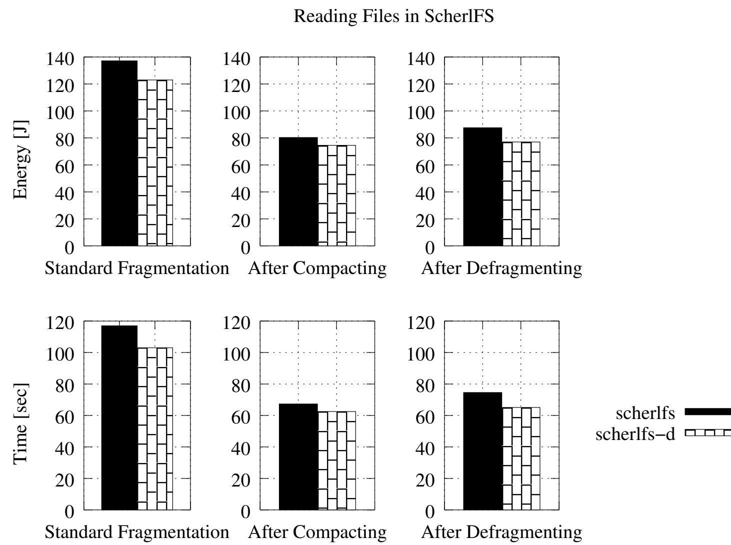

Figure 7.6: Impact of cleaning modes on ScherlFS read performance (part 1)

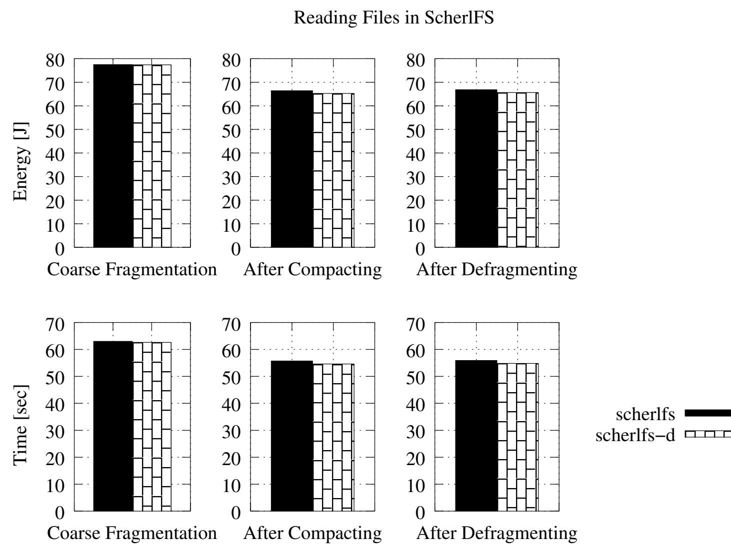

Figure 7.7: Impact of cleaning modes on ScherlFS read performance (part 2)

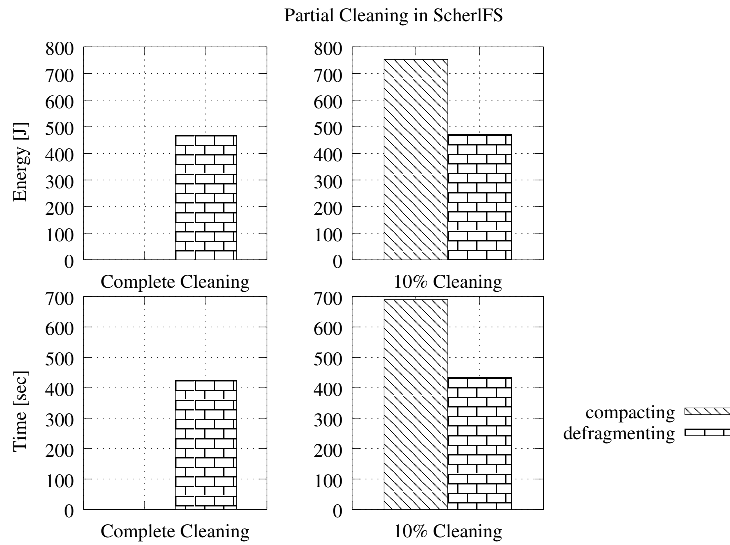

Figure 7.8: Partial application of the cleaning process

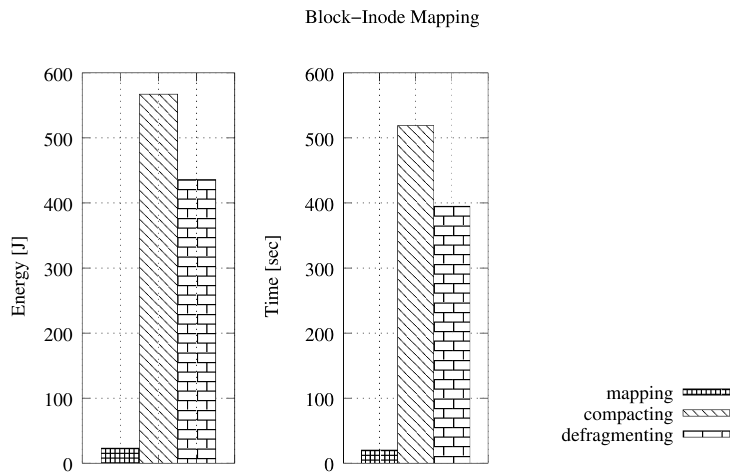

Figure 7.9: Energy consumption and time needed for creating block-inode mapping

Chapter 8

Conclusion

The most important conclusion is that the Linux buffer cache can tremendously speed up operations for all file systems.

In the course of this work the concept of log-structured file systems has be explained and the advantages and disadvantages have been outlined. The reduced number of disk seeks and the sequential data layout allow for energy efficient read and write operations and therefore increased idle times which in turn favor the low power modes of modern hard drives. The need for an explicit free space management for continuous operation and the resulting energy costs are the main drawback of a log-structured file system in regard to power consumption. Various free space management strategies have been described in general and some of them for an existing log-structured file system called ScherlFS implemented by Holger Scherl for his thesis[2]. Two of these free space management strategies have been implemented as prototypes for the ScherlFS file system and tested in different situations.

The main drawback of the two copying solutions is that they require large amounts of energy during operation. They also need exclusive access to the file system and therefore are unsuitable for incremental free space management during runtime. While this is already an important reason to consider other free space management strategies, the measurements in this paper showed that even the two similar compacting and defragmenting approaches can exhibit vastly different performance characteristics and other influences on the operation of a log-structured file system. For example, the defragmenting cleaner prototype performs much better when the whole log needs to be cleaned. While the resulting better layout of the data should improve read operations compared to the compacting cleaner, this assumption could not be proved by the tests in this thesis. However the read performance is significantly improved for read accesses in comparison to a fragmented log. On the other hand, the compacting cleaner performs better if only a small portion of the log needs cleaning and, in contrast to the current defragmenting cleaner implementation, it is nearly independent of the number of files on the ScherlFS device. If enough energy is available,

e.g. during recharging of a mobile device, defragmenting or compacting can boost the performance of future read operations while the current energy consumption is not important anyway.

Thus, for energy efficient operation of a log-structured file system in general and ScherlFS in particular a combination of some complementing free space management strategies is advisable. The file system should choose the most appropriate cleaning approach depending on the current situation. Compacting and defragmenting cleaning is best used when enough power is available, as they influence the operations of the log-structured file system in a positive way and help save energy later, while the current increase in energy consumption does not matter.

However, the question remains whether the two cleaning prototypes are suitable to allow ScherlFS to operate more energy efficiently than other available file systems. Obviously, this depends on the overall usage of the storage medium: If there are mainly new file creations and data reads, and few modifications, the compact data layout can compensate for the increased energy cost during cleaning. However, if there are many small updates, the compact data layout will not last long enough to make up for the cleaning overhead. A possible way to tell definitely whether the ScherlFS file system in combination with the cleaner prototypes is suitable for a specific situation is to analyze traces of I/O patterns of the application in question.

Bibliography

[1] Hitachi Global Storage Technologies. Hitachi familiy of microdrives datasheet.

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

[3] John Zedlewski, Sumeet Sobti, Nitin Garg, Fengzhou Zheng, Arvind Krishnamurthy, and Randolph Y. Wang. Modeling hard-disk power consumption, March 2003. Department of Computer Science, Princeton University.

[4] IBM Coporation Storage Systems Division. Adaptive power management for mobile hard drives, January 1999.

[5] Paul M. Greenawalt. Modeling power management for hard disks. In *MASCOTS*, pages 62–66, 1994.

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

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

[8] 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 *Proceedings of the sixteenth ACM symposium on Operating systems principles*. ACM Press, 1997.

[9] Jacques Cohen and Alexandru Nicolau. Comparison of compacting algorithms for garbage collection. *ACM Trans. Program. Lang. Syst.*, 5(4), 1983.

[10] Mark S. Johnstone and Paul R. Wilson. The memory fragmentation problem: solved? *ACM SIGPLAN Notices*, 34(3), 1999.

List of Figures

2.1 Hard drive geometry 3

4.1 The ScherlFS superblock structure on disk 11
4.2 A ScherlFS inode structure on disk 12
4.3 A ScherlFS extent for file/directory inodes on disk 12

5.1 Compacting the log 16
5.2 Defragmenting the log 16
5.3 Hole-plugging 17
5.4 Hole-plugging with data chunk splitting 17
5.5 Threading approach - four new data blocks are appended to a nearly full log 18

6.1 Example for source and destination regions in defragmenting mode 23
6.2 Regions shown in figure 6.1 after defragmentation 25

7.1 Characteristics of the different fragmentation samples 30
7.2 Energy consumption and time needed for creating new files . 32
7.3 Energy consumption and time needed for updating files in-place 32
7.4 Performance characteristics for reading 33
7.5 Performance of the two cleaning modes 34
7.6 Impact of cleaning modes on ScherlFS read performance (part 1) 35
7.7 Impact of cleaning modes on ScherlFS read performance (part 2) 35
7.8 Partial application of the cleaning process 36
7.9 Energy consumption and time needed for creating block-inode mapping 36

Reorganisation in energiebewussten Dateisystemen

Im Laufe der letzten Jahre haben sich mobile Computer und Embedded Systems immer mehr verbreitet. Der Hauptvorteil von mobilen Geräten für die Benutzer ist die größere Flexibilität, da man nicht auf eine stationäre Stromversorgung angewiesen ist. Durch den technologischen Fortschritt werden komfortablere und komplexere Anwendungen möglich, während gleichzeitig der Bedarf an solchen Anwendungen steigt. Viele dieser Anwendungen benötigen eine Form von Datenspeicherung. Im gleichen Maße wie die Anzahl der Features von mobilen Geräten wie Handys, Digitalkameras oder PDAs mit mobilen Büro-Anwendungen steigt auch der Energiebedarf. Auch wenn zahlreiche und wichtige Fortschritte im Bereich des Energieverbrauchs, und damit der Laufzeit im Batteriebetriebs gemacht wurden, ist bisher ein möglicher Bereich zur Reduzierung des Energieverbrauchs weitgehend vernachlässigt worden: Speichermedien. Eine Möglichkeit die Energie-Effizienz von Speichermedien zu steigern, ohne neue Materialien für die Datenspeicherung zu erforschen oder neue Hardware zu bauen, ist die Anordnung und Zugriffsmuster der Daten auf handelsüblichen Festplatten wie zum Beispiel des Hitachi Microdrives zu modifizieren.

In seiner Studienarbeit hat Holger Scherl eine alternative Möglichkeit untersucht und implementiert, die die Daten auf der Festplatte zu organisieren und dadurch den Energieverbrauch durch Disk Seeks und Rotationsverzögerungen zu minimieren: so genannte Log-Structured Dateisysteme. Obwohl der Prototyp und die bisherigen Energiemessungen sehr ermutigend sind, fehlt der Implementierung noch ein wichtiger Bestandteil, welcher eine längerdauernde Benutzung des ScherlFS Dateisystems ermöglichen kann. Ein Log-Structured Dateisystem spart Energie zu Lasten von temporär verschwendetem Speicherplatz, daher ist eine effektive Freispeicherverwaltung nötig, die die unbenutzten Speicherregionen wieder verwendbar machen kann.

In dieser Ausarbeitung werden verschiedene Möglichkeiten der Freispeicherverwaltung für Log-Structured Dateisysteme untersucht, sowie die Effekte dieser verschiedenen Ansätze auf die Performanz und das Energieverhalten des Dateisystems analysiert.

Im Rahmen der Studienarbeit wurden zwei Ansätze der Freispeicherverwaltung für das Log-Structured Dateisystem ScherlFS implementiert und in Bezug auf Energieverbrauch und Performanz unter verschiedenen Bedingungen getestet: eine defragmentierende und eine kompaktierende Variante. Obwohl beide Verfahren sehr ähnlich sind zeigen sich dennoch zum Teil gravierende Unterschiede im Laufzeitverhalten und Energieverbrauch in bestimmten Situationen. Die kompaktierende Variante skaliert besser mit steigender Anzahl von Dateien und arbeitet besser für kleine Bereiche im Log, während der defragmentierende Prototyp eine deutlich bessere Performanz zeigt, wenn der freie Speicher im ganzen Log zurückgewonnen werden soll. Der Einsatz einer kompaktierenden oder defragmentierenden Strategie zur Freispeicherverwaltung hat einen positiven Effekt auf zukünftige Lesezugriffe auf das Dateisystem. Da der Energieverbrauch der verschiedenen Verfahren zur Freispeicherverwaltung von verschiedenen Faktoren wie der Fragmentierung, der Häufigkeit der Zugriffe und dem Zugriffsmuster abhängt, ist ein adaptiver Ansatz als Kombination verschiedener Strategien denkbar.