Design und Implementierung eines energiesparenden Dateisystems

Studienarbeit im Fach Informatik

vorgelegt von
Holger Scherl
geboren am 10. Mai 1977

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

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

Beginn der Arbeit:	6. August 2003
Abgabedatum:	22. Januar 2004

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, 22. Januar 2004

Design and Implementation of an Energy-Aware File System

Studienarbeit

by
Holger Scherl
born May 10th, 1977, in Forchheim

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

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

Begin: August 6th, 2003
Submission: January 22nd, 2004

Abstract

The development of small mobile hard disks with large storage capacities has changed the application requirements of modern mobile computers and embedded systems significantly. While these devices are mostly battery-powered, all usable power saving techniques have to be integrated to prolong their up-time.

Most mobile devices suffer from the restriction of small sized memories. It is common that the available memory is exclusively used by the operating system and the running applications. Therefore, in contrast to normal workstation computers, the caching of disk blocks at the operating system level is problematic and not as effective for these devices due to the large memory requirements of that technique. However, existing file systems organize file data and meta data to match performance requirements by relying on the presence of a considerable amount of system memory for caching purposes. As far as hard disks are concerned, valuable battery power can be saved by avoiding disk seeks and introduced latencies due to rotational delays.

This work examines the effects of different file system layouts on hard disk energy consumption. It is shown that energy efficiency of file systems is heavily influenced by the used data layout and file organization principle when disk caches are mostly not available. Guidelines for a low power file system design are developed. A new implementation of a file system for the Linux kernel based on these recommendations is provided to show that energy efficiency is significantly improvable by adapting the file system layout in systems which do not have the possibility to use disk block caching techniques.

Contents

Chapter 1

Introduction

In the last few years mobile devices and embedded systems have set one major trend in the computer industry. Independence of an external power supply for a reasonable time period is a crucial necessity for mobile computers that deserve their name. For this reason energy consumption has been recognized as a major challenge of mobile system design.

At the same time the miniaturization of large storage devices has widened the possible range of mobile applications and therefore their purpose of utility for the end user. Today mobile web browsers, audio-video players and recorders, integrated mobile phones with lots of additional features and small office suites are only a small subset of the possible uses for such devices. As mass storage media gain more and more importance for the applications of modern mobile computers, such systems are often equipped with mobile hard disks as alternative means to flash memory. Recently, the IBM or now Hitachi Microdrive [27] has mostly been used in these devices for secondary storage. Regarding energy consumption substantial savings became possible in other components but not as much in the area of hard disk power [13]. That's why in comparison to earlier models of portable computers, modern devices spend a greater percentage of their power consumption on the hard disk. Thus, power saving techniques for mobile hard disks offer high potentials to prolong system up-times.

Nowadays hard disks can improve their energy efficiency significantly by exploiting their low power modes. As the transition between such modes consumes a remarkable amount of energy and introduces latency issues a notable increase in the length of idle periods is necessary to save energy. The research community is very active in this area, yet this technique either requires large system memories or an application involvement in order to be most efficient.

Similarly, as far as performance and energy efficiency is concerned, existing file system technology is designed to rely heavily on the availability of a large amount of memory to dynamically hold frequently used data in disk caches. However, due to reasons of mobility and energy efficiency, one characteristic trait of mobile hardware design is a need for miniature components implying the consequence that their performance and equipment cannot reach the outcomes of today's workstations. Generally mobile devices suffer from lower processor speed and small sized memory. There the available memory is mostly used by the running applications and the operating system itself. Therefore, the memory needed for disk block caching is not available and the energy requirements of the file system become most sensitive to the data organization on disk because many disk seeks and rotational latency delays are required by reading and updating file system internal data structures which only manage storage layout. Therefore a low power file system design has to eliminate this wastage of energy as much as possible. When only powered with batteries, a file system could easily adapt to this pretension by organizing its data in a non optimized manner but with lowest energy demands. While users are always obliged to charge their batteries mobile computers have to be connected to an external power supply from time to time. Then energy efficiency is of minor importance, providing the chance to clean up imperfect data organizations and initiate defragmentation techniques.

The present paper will focus on adapting the file system layout to save hard disk power in an environment with small system memory.

The rest of this work is structured as follows.

The second chapter gives an overview of the research areas of previous work. Chapter 3 deals with techniques that are used to reduce hard disk energy consumption in the lowest levels of the operating system and the apparent drawbacks of existing file system technology. Their effects on energy consumption in mobile systems are described. A new approach to incorporate energy efficiency into the file system layer is presented in chapter 4. Chapter 5 further describes the implementation of a prototype file system which is based on the Linux 2.4.21 kernel. Chapter 6 reports the energy saving results related to existing Linux file systems using real world applications and synthetic tests as an evaluation basis. Finally, hints for possible future improvements are listed in chapter 7.

An energy-aware file system as suggested in this paper may be embedded in

an overall hard disk energy-saving concept, for example by merging in the concept of cooperative I/O as introduced by Weissel et al. [29]. When large memories are available for the caching of disk blocks the proposed file system design could be optimized to collaborate with the updating and prefetching methods as suggested by Papathanasiou et al. [21] to reach a ”bursty” access pattern on the hard disk, thus increasing idle times and energy efficiency.

Chapter 2

Related Work

2.1 Power Management

Energy-aware operating system. During the last few years design paradigms in the area of power-conscious operating systems have incorporated energy as a first class system resource. The main target of such systems is the control of energy consumption and battery lifetime by unifying resource management and by introducing energy accounting mechanisms. These approaches are explicitly designed to be independent of application software preventing the need of rewriting or adapting user programs.

The importance of energy efficiency as a primary metric in operating system design is emphasized by Ellis et al. [5], [28]. ECOSystem [32] is a modified Linux in which a fair allocation of energy among competing applications is accomplished. It aims to provide system wide energy management under the control of user preferences. In a more recent report [31] an increased energy efficiency of the hard disk is realized through the coordination of disk accesses. This is achieved through bidding and pricing techniques that are based on the ”currentcy” metric introduced in ECOSystem.

Application adaption. On the other hand the approach of Flinn et al. [6], [20] enables applications to reduce their energy requirements by dynamically modifying their behavior. Monitor techniques for energy supply and demand are performed at operating system level to guide running applications via special system up-calls. Then applications can be rewritten to select an acceptable tradeoff between energy consumption and application quality by using the available information.

Device level power management. To save energy industrial efforts have been concentrated to integrate low power modes into modern mobile devices and their components. These modes are associated with a different level of power consumption and response time. In the area of power management for hard disks heuristic transition policies emerged both at hardware and operating system level. They can be divided into algorithms with fixed or adaptive time-out and predictive policies. Different shutdown policies have been suggested by [10], [3], [11], [12], [14], [7]. A detailed comparison and evaluation of several algorithms can be found in [4] [17]. Different commercial solutions which address OS-level power management are the Advanced Configuration and Power Interface (ACPI) [1] and Microsoft's OnNow [18].

Cooperation between applications and operating system. Heath et al. investigated the potential benefits of application supported device management for optimizing energy and performance [8]. To increase disk idle times they propose that applications should explicitly cluster read operations together and inform the operating system about upcoming idle intervals. Furthermore, a compiler framework that is able to perform the transformations mentioned is given in [9]. Lu et al. argue for a special system call that details device usage requirements and future access patterns [15]. Thereby applications can inform the operating system about future idle times to perform energy efficient shutdown decisions of devices. The given information is further exploited by intelligently reordering the schedule of processes to achieve a clustering of device requests which results in the prolongation of idle periods. A task based approach without application involvement is suggested by Lu et al. [16] by computing device and processor utilizations of each task at the operating system level. The shut down of devices is performed when having a low overall utilization. By reshaping disk usage patterns Weissel et al. [29] achieve higher energy efficiency by increased idle periods in the ,,cooperative I/O" project. They provide an API that allows processes to pass the urgency (delay time) of each individual read/write operation whereby requests from different processes can be grouped together and device activations are prevented by defining requests as deferable or abortable. Papathanasiou et al. [21] also maximize idle periods to gain higher energy efficiency. Therefore to achieve a bursty access pattern they altered the linux caching policies with aggressive prefetching and buffering techniques. Application support for write-behind caching is possible by a new flag in the open system call.

2.2 Log-structured File Systems

The principles of log-structured file systems (LFS) were introduced in 1991 by Rosenblum et al. [22]. They described a new technique for disk management that aimed to speed up both disk writes and recovery times and presented a prototype implementation for the Sprite operating system. Seltzer et al. [24] 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 [25]. 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. tried to implement a log-structured file system in Linux 2.0.X series by adding a logging-layer between an adapted version of the ext2 filesystem and the block device [2]. Matthews et al. described possibilities to improve the performance of log-structured file systems with adaptive methods for a wider range of workloads [19]. Layout adaption to match hardware characteristics of storage devices through track-aligned extends is performed by Schindler et al. [23]. By utilizing disk-specific knowledge they were able to increase IO efficiency by reducing head switches for mid to large requests. They suggest matching segments to track boundaries to reach a better performance both for cleaning and writing in log-structured file systems.

Chapter 3

Motivation

3.1 Magnetic Disk Characteristics

Differences in disk hardware are hidden to the storage manager by introducing the abstraction of logical block addresses (LBA). That enables file systems to consider disk storage as an array of equally sized blocks in a sequentially numbered order. Before discussing strategies that are able to increase energy efficiency of hard disks it is necessary to thoroughly understand beneath that abstraction the characteristics of magnetic disks that influence both their performance and energy consumption.

After describing the functionality of hard disks in general, the characterizing parameters are exemplified with the Hitachi Microdrive [27]. Then commonly used optimization techniques to overcome performance losses of hard disks performed at the lowest levels are briefly described.

3.1.1 Anatomy of a Disk Request

Data storage on hard disks is divided into sectors that are usually 512 bytes in size. These are organized in a stack of rotating platters and are accessed by various read/write heads together mounted on a moveable disk arm. For one disk arm position the heads can access a certain amount of sectors which are arranged as a circular ring on each platter. In this position sectors that are accessible by a single head together are called a track whereas a cylinder contains all tracks for one disk arm placement. LBAs are mapped to physical locations on disk by translating the logical address into cylinder/head/sector (CHS) combinations. Then the hard disk satisfies requests by switching to a sequence of different operating modes. If the requested sector lies on a cylinder which is different to the present disk arm

position first the seeking mode has to be entered. When the disk arm has arrived at the correct location the disk selects the appropriate head and steps into rotation mode to wait until the specified sector rotates under the head. Then the reading or writing mode is responsible for the actual data transfer. The time spent in each of these modes is referred to as seek time, rotational latency and transfer time.

Concerning the Microdrive, the average time spent in seeking mode adds up to 12 milliseconds. Rotational latency is determined by the speed of the rotating platters. Then a rotation speed of 3600 RPM for that drive corresponds to 8.33 milliseconds on average. Using the power measurements of Zedlewski et al. [26] it becomes possible to calculate energy overheads for the Microdrive according to the different operating modes. In Table 3.1 the average consumed powers according to the various modes and the overheads for single operations are listed. Due to the fact that seek operations and rotational delay spend much more energy than the data transfer of single blocks, their avoidance is a jutting candidate for improving energy efficiency.

Mode	mW	mJ
Seeking	637	7.64
Rotation	594	4.95
Reading	627	0.19
Writing	756	0.23
Idle	222	-
Stand-by	61	-

Table 3.1: Average power consumed by the IBM Microdrive. The right column indicates the typical energy overhead for one operation in average (seek, rotational delay, 1 KB read, 1 KB write).

3.1.2 Low Level Optimization techniques

There are two commonly used optimization techniques concerning the negative effects of rotational latency. Some hard drives offer a feature called zero-latency access. After a seek their firmware immediately reads all sectors of a cylinder into an internal buffer in arbitrary order, thus anticipating additional latencies for subsequent accesses to the same cylinder at hardware level. Some drives also offer integrated write caches to eliminate rotational latency as seen from the operating system. However, the write operation itself still suffers from latency issues. If more than one disk request is to be queued, the operating system performs a technique

called block clustering. Requests of adjacent sectors are joined together to form one big request. Therefore the latency can be shared among them.

To avoid seeks request scheduling is used at the lowest operating system level. By this technique an elevator-algorithm is performed which orders remaining requests with regard to their sector numbers. Therefore a reduction in seek time is achieved by using the overall minimal seek distance.

3.2 Linux File Systems

All techniques described so far are performed at very low levels but at best they are able to further support file systems to reduce rotational latencies and disk seek times. Above these methods file system technology is principally responsible for avoiding disk seeks and reducing rotational latencies. Modern file system designs try to attend to this duty by optimizing data layout on disk. However, they are designed to rely heavily on large memories to prove performant and energy efficient. In this section I briefly describe existing file system designs and outline their drawbacks with regard to energy efficiency in environments with small sized memories.

The second extended file system (ext2fs) is the most well known file system design in the Linux community. It is famous for its stability and good performance. Files are represented by a structure, the so called inode, which maintains meta data and pointers to the file's data blocks. By saving inodes and their data blocks into certain regions on disk, the block groups, fragmentation issues are addressed. All block groups in the file system have the same size and are stored sequentially. Block groups are described by group descriptors which provide information about used blocks, inodes and directories. There are bitmap fields in each block group to specify which inodes and data blocks are free or used within a group. Operations like creating a file or appending data to it require an update of these data structures. To prevent the file system from permanently accessing the same disk blocks, caching of these structures is introduced. Group descriptors are permanently cached and at least eight blocks are cached, which is realized by a LRU mechanism. When created each data block remains at the same physical position on disk over the whole file life span. While the file system fills up most data blocks of new files will be created in a fragmented manner.

In the worst case these data blocks are located in different block groups, which requires additional disk seeks for sequential file accesses. In addition, disk seeks caused by meta data accesses will become very likely, especially in systems which have not much system memory. Due to the fact that disk seeks are a waste of energy a low power file system must develop a different design.

In case of power failure or crash the ext2 file system has to run an exhaustive scan and sanity check on the meta data both of the block groups and of the inodes and their data in order to ensure file system consistency. As this is a very power intensive task a low power file system must use other strategies for this problem.

Journaling file systems overcome the problem described above by inheriting database transaction and recover technologies into the file system. Meta data modifications are logged into a reserved area of the file system, the journal, before updating the affected disk blocks. After a crash only the portion of files which have an entry in the journal have to undergo the consistency check.

In fact the ext3 file system is not different from an Ext2fs despite an added journal file. Unfortunately updating the journal on every meta-data change induces a need for more seeks and thus a power wastage in normal operation mode.

The Reiser file system is also a journaling file system based on fast balanced trees (B+ Trees) to organize file system objects. Small files are saved directly into a tree node to avoid disk seeks. It also tries to store file information closely to file data.

Another log-based file system is developed by IBM, which is currently used in high performance e-business file servers. Again, only meta-data changes are logged. The logging style introduces a synchronous write to the log disk into each operation that changes meta-data structures. This performance cost is reduced in server systems by group commits, which combine multiple synchronous write operations into a single write operation. Asynchronous logging schemes are also able to decrease the performance losses.

Again it is very likely that such sophisticated techniques behave well only in environments with lots of memory for caching the data structures. Thus it is expected that journaling file systems lead to lots of disk seeks in systems that cannot cache these structures. Moreover, further disk seeks are also required by the log-based techniques themselves.

The FAT32 file system on the other hand is very simple in its nature. The superblock contains the block number of the root directory. Directories store the meta data of files together with their first data block. Further data blocks of directories and files can be looked up in the file allocation table (FAT) at the beginning of the partition. The FAT is nothing else than an array of 32 bit integers. The position within that array corresponds to a disk block number whereas the stored value indicates the next block in that file. Accesses to files always end up with hopping around in the FAT. For a typical Microdrive with 1 GB capacity and a block size of 1 KB the FAT would contain 4 MB of disk space. As this amount of data is not cache-able it ends up with a large amount of disk accesses and disk seeks.

Chapter 4

Design

In contrast to normal workstation computers mobile devices have to get by with a limited power supply. The benefit of such systems closely relates to the time period in which they can be powered with their batteries. Therefore prolonging the up-time of mobile systems is the most important point to deal with. Design issues for energy conscious file systems are examined. The most common drawbacks of existing file system designs can be avoided with regard to energy consumption.

First guidelines for an energy efficient file system design are proposed. Then ideas to fulfill many of the given objectives are presented.

4.1 Guidelines for Energy Efficient File System Design

When accessing a file on disk several steps are required to fulfill the request. When a file name is given, the corresponding information which describes the file contents and block locations has to be identified and read in. Only then can file blocks be selected for transfer. When new parts of a file are going to be written to disk, free places on disk have to be found where the data can be written to. Then meta data structures related to the file must be updated.

As outlined in Chapter 3 disk seeks and rotational latencies require a significant amount of energy. For example, with the energy which is necessary to perform an average disk seek accompanied by an average rotational delay 66 KB of data can be read from an IBM Microdrive hard disk. By servicing hard disk requests the energy of the data transfer itself has to be invested. Therefore, the avoidance of latencies caused by disk seeks and rotational delays is the only way to improve energy efficiency. The problem of reducing energy consumption of the hard disk

becomes equivalent to the problem of optimizing the performance of the disk requests, which is very difficult to realize in systems without large disk caches.

Therefore, a low power file system design can incorporate most of the following guidelines to gain energy savings in battery powered mode.

4.1.1 Reduction of Meta Data Updates

When new data is written to disk or existing data is deleted the disk manager has to update its meta data structures. These are needed to manage file system dependent information and to identify file locations and free spaces on disk both for data blocks and file information structures. When updating or creating file data meta data structures also have to be cared for. This causes seeks and rotational latency delays when the affected structures and the positions for reading and writing are not located at near physical disk positions. Therefore meta data updates should be kept minimal, ideally bundled together with the executing disk accesses and performed at the disk location which was already affected by the causing access.

4.1.2 Sequential Arrangement of Meta Data and Data Blocks

Despite the data contained in a file several information about the file itself has to be maintained by the disk manager. The size of the file, its type and the location of the file blocks on disk are a few examples. This information is generally associated with meta data. Therefore two different entities have to be saved to disk: meta data and file data. To fulfill file requests such as read and write operations it is necessary to access and update the meta data of the corresponding file. If meta data and file data is organized at physically distinct disk locations it becomes necessary to issue two disk requests accompanied by two seeks and two rotational latencies when a task accesses file data. What is most important for workloads containing many small files, it is possible to avoid the overhead of one seek and one rotational latency by aligning meta data information and the data blocks successively on disk. Unfortunately, as opposed to a fixed positioning dynamic positions 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.

4.1.3 Sequential Reading and Writing Behavior

To enable energy savings for workloads which are characterized by sequential file accesses data blocks of files can be clustered together at near physical disk positions. Then sequential file accesses are able to avoid many seeks and rotational latencies. Therefore, all blocks of a file should be grouped into one large extent on disk together with their meta data. However, it is very difficult to attain this ideal behavior when the file system becomes fragmented. Update-in-place file systems try to approximate an ideal sequential data layout only at the creation time of the file data blocks. If the file system is heavily fragmented at this point file access performance will suffer from an imperfect data layout over the whole file life span. However, it is possible to make the disk manager capable of automatically creating unfragmented empty space on disk. This makes it possible to arrange the file data in a sequential manner. However, the task of free space generation requires energy, which has to be invested for that purpose.

4.1.4 File System Reorganization

When small files are often accessed in the same sequence these files could also be arranged sequentially to save energy. Figure 4.1 shows the perfect organization of five files. The arrows indicate the access order of the files. For example file one is always accessed right before file two. As described above, data reorganization is already used to generate large extents of empty space. This technique can be extended to match file organizations on disk to read access patterns. This approach is most beneficial for workloads which access many small files. Then additional

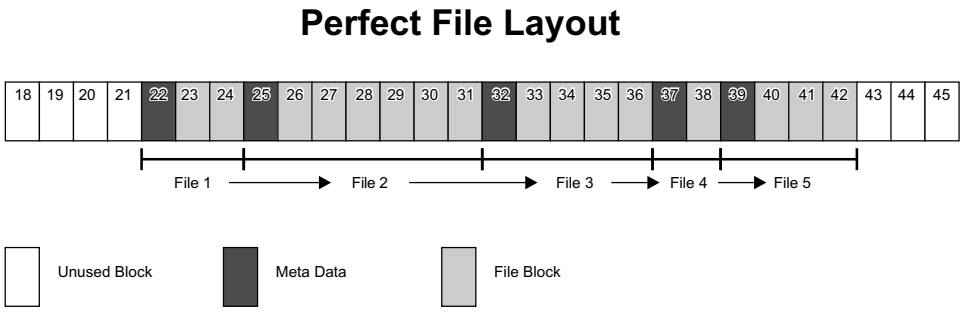

Figure 4.1: Perfect file system layout

energy savings are possible by avoiding disk seeks and rotational latency delays between file accesses.

This approach is based on the hypothesis that files can be grouped to semantical units which are mostly accessed together. Two problems have to be addressed. First the mentioned file groups have to be determined. This can be done by statically introducing relationship constraints. For example files, which are residing in the same directory, can be assigned to one of these groups. Another solution to the problem can be approximated by observing file access patterns. Then groups can be found by the assumption that past file accesses are a good predictor of future access patterns. However, this method introduces overheads for both saving the observation results and the computation of the groups. The computation can be done by creating an access graph from the observations and solving the graph partitioning problem for it. Although this problem is NP-complete many heuristic solutions exist in the literature. Once the disk manager knows the file groups which should be arranged together, the differences between the current and desired layout have to be analyzed and if necessary I/O requests have to be issued.

4.1.5 Adaptive Energy Use

As users are obliged to charge batteries mobile devices are attached to external power supplies from time to time. Then energy consumption is mostly of no concern. However, when powered by batteries, every Joule is precious. Therefore it is expedient to differentiate two operating modes: battery and powered mode. In battery mode the system is disconnected from external power. There the most important objective is to get by with the remaining battery resources. I define powered mode by being attached to an external power supply. Here energy consumption is not so much important for the users of mobile systems. In battery mode file system operations should be performed with lowest energy requirements. For the short term accessing and updating data on disk can be done in a non-optimized manner to save energy. Data organizations can be cleaned and optimized later when changing to powered mode. There the workload of the device is determined by long idle periods which can be used for the cleaning operations. Therefore a different file system behavior depending on the operating mode of the system enables significant energy savings in battery mode by writing to disk with lowest energy requirements. On the other hand only this distinction of operating characteristics justifies to invest energy in powered mode for empty space generation and data reorganization as outlined in Sections 4.1.3 and 4.1.4.

4.1.6 Fast Crash Recovery

Mobile computers are often used until the battery power is consumed completely. Furthermore unclean shutdowns are often carried out by the users of these devices. But file systems are usually left in an inconsistent state after a crash due to power or software failures and improper shutdown actions. Therefore consistency checks have to be performed to ensure data reliability. A complete consistency check requires scans over the entire disk to rebuild and verify all data structures of the file system. This has enormous overheads both in time and power.

Thus, it appears that fast recovery techniques should be incorporated in a modern file system design. This implies the integration of a technique that is able to identify the last modified data on disk. Then the performed recovery algorithm has to operate only on a fraction of the whole disk. Therefore disk operations are recorded in a log structure before their execution. After a crash the log still contains the locations which are influenced by the last disk requests. However, managing and updating the log is accompanied with further overheads in energy usage for all file system operations which will alter meta data on disk. A low power file system has to update its log structure with minimal overheads in energy to justify the incorporation of recovery techniques.

4.1.7 Prolongation of Idle Periods

Finally, energy efficiency of mobile hard disks can be improved significantly by exploiting their low power modes. However, transitioning between these modes introduces a higher energy consumption for a short term depending on the hard drive used. This increased power usage has to be amortized by resting for a sufficient amount of time in low power mode. It is referred to as the so called break-even period. Therefore lengthy idle periods are another goal for low power file systems to achieve. There are two approaches which can be used. First, accesses to hard drives can be avoided when resting in a low power mode by an application involvement introducing deferrable and even abort-able read and write system calls. Second, with large system memories both data for reading and writing purposes can be saved into operating system caches. Then a prolongation of idle times can be accomplished by incorporating aggressive prefetching and write-behind caching in the buffer cache mechanisms of mobile operating systems when the policies to update dirty disk blocks are changed to a ”bursty” pattern.

4.2 Log-structured Approach

To stick to the proposed guidelines for a low power file system design I argue for a log-structured approach similar to the ideas of Rosenblum et al. [22]. A log-structured file system logically considers the disk as an infinite append-only log. In contrast to other file system designs, meta data of files, their indexing information and data blocks are not written to static locations on disk but are simply appended to the log. This gives rise to achieve a successive alignment of meta data and its appendant data blocks at physical block positions on disk. Furthermore, as data blocks are also appended to the log this results in their successive disposition. It renders possible the feasibility of a sequential reading and writing behavior with almost no administrative overheads caused by updates of file system specific internal data structures.

As fast recovery techniques have to be incorporated in modern file system designs this approach can intuitively perceive this pretension by the append only log behavior. A log-structured storage manager can easily provide periodic checkpoints which allow recovery to proceed efficiently from the most recent checkpoint to the tail of the log.

Although the basic idea of this approach is very simple, two key issues have to be resolved to achieve the expected benefits. First the retrieval of information from the log is more complex as compared to indexing into traditional file systems. While section 4.2.1 covers that problem there is a second issue that has to be resolved. The disk manager has to manage free space on disk. For accomplishing this more difficult task different strategies are possible depending on battery or powered mode. This is the topic of Section 4.2.2.

4.2.1 File Location and Reading

Although a log-structured approach might suggest that sequential scans are required to retrieve information from disk, this can be avoided by integrating index structures to the log, which enables random access retrieval. In the Unix world disk addresses of file blocks and information about files are stored within inode structures. For each file a simple address calculation yields the disk address of the corresponding inode since inode structures are stored at fixed locations. In contrast, in log-structured file systems inodes are written to the log as well, which results in a dynamic position on disk. To maintain the current location of each inode a new data structure is introduced, the so-called inode map. If the identifying number for

a file is given the inode map can be indexed to determine the proper disk address. Since the inode map is divided into blocks it can be written to the log as usual. Normally the inode map is compact enough to keep its active portion cached in main memory. Therefore inode map lookups rarely require disk accesses and for that reason additional disk seeks.

4.2.2 Free Space Management

Initially a log-structured file system maintains all free space in a single extent. But by the time the disk fills up with new log writes, free space must be generated to satisfy more write requests. Fortunately, not everything which is part of the log corresponds to recent versions of files. When updated data is written to the end of the log, a previous copy of the data on disk still remains at the old location. Unless the file system has to support undo requirements of file changes, these data blocks can be considered as dead space or a hole in the log, thus free space will have been fragmented into many small extents. Therefore, for servicing additional updates and writes free space management becomes an essential premise. The goal is to provide and identify regions of free space on disk, which should be organized preferably in large contiguous extents. There are three possibilities to attain the goal:

- The log can be threaded around the still live blocks.
- Live data can be copied out of the way.
- A combined threading/copying approach.

4.2.2.1 Threading

The threading approach leaves live blocks at their original position on disk (figure 4.2). New log blocks are written to the holes of the log that have emerged at positions where previous incarnations of now updated or deleted blocks resided before. It must be possible to identify these dead places when using this technique. Therefore it is required to insert dead blocks into a list to mark free positions of the log. In case of a crash this list is also used by the applied recovery technique to identify the log regions. It is possible to manage the free block list with minimal overhead. When writing new data to disk, old log blocks are simply inserted into the free block list which is partially cached in memory. A compact representation can be achieved by organizing data blocks as extents which indicate a starting block

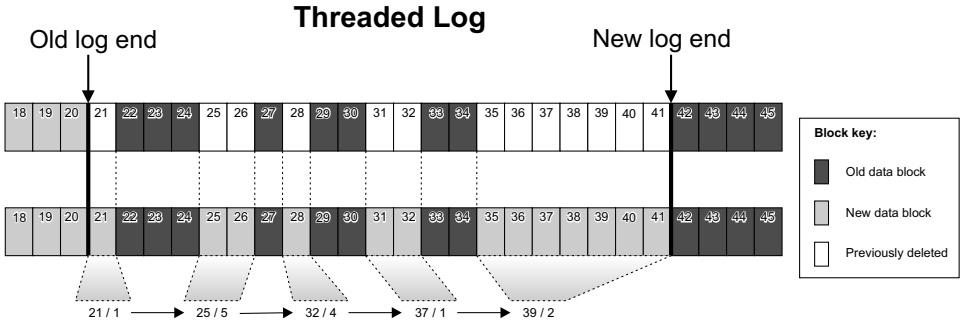

Figure 4.2: The threading technique. Log ends before and after a log wrap are shown. The extent structures which indicate dead log space are marked.

number and the number of contiguous free blocks after that position. When the size of the cached list entries exceeds a configurable amount of space, the cached data is simply written to the log together with other log writes, thus the free block list can be treated by the implementation as nothing else than a special file. Now when the log is filled up and new log space has to be generated this list can be used to compute a new threaded log region. For efficiency reasons the threaded log has to be generated carefully. This requires to sort the extents by their starting block number. As well joins of extents have to be accomplished when the starting or ending block number coincides with the starting block number of an already existing extent.

The threading technique is able to operate at a high disk capacity utilization and with low overheads for free space management. Thus, this approach is well suited for battery mode operation because the expensive copying of live data is not required as opposed to the techniques described later in sections 4.2.2.2 - 4.2.2.4. For the short term threading operates at the lowest energy requirements in comparison to the other techniques. This is achieved by the price of imperfect data organizations. Free blocks are fragmented between the still live blocks. As the degree of fragmentation depends only upon the workload the disk manager itself has no influence to control fragmentation issues while it is not possible to prevent the free space on disk to become fragmented. Under more unfavorable workloads large sequential transfers become not possible any more and the time spent striping over survived blocks drops disk access performance especially at a high disk capacity utilization. But when changing to the powered operating mode other free space management techniques are well suited to overcome the fragmentation issue

by the prize of a higher energy usage. Fortunately, energy consumption and disk utilization are not of much concern then. Therefore additional techniques have to be incorporated into the design for free space management in the other modes.

4.2.2.2 Copying

As an alternative to skipping around the still live blocks after a log wrap it is possible to move the live blocks somewhere else as it is shown in figure 4.3. This allows the log to be written to the cleared locations. New large extents of data will become available but beforehand energy and time has to be invested by the copying operations. This happens at each log wrap, which creates the need to copy around long lived data blocks multiple times, which constitutes an enormous energy wastage. However, this can be justified in powered mode if energy efficiency increases in battery mode operation.

Although new large extents of empty space are created, the goals as outlined in Sections 4.1.2 and 4.1.4 are not achieved by the pure copying approach. However, the technique can be extended to eliminate the fragmentation of file data blocks and also to reorganize files in groups of physically near disk blocks. Data reorganization and defragmentation techniques are well suited to be integrated into the cleaning algorithm because many of the affected disk blocks are already read in and saved to other locations on disk.

4.2.2.3 Hole-plugging

It is more difficult to realize the copying approach at a high disk capacity utilization because many data blocks have to be copied around before new large extents of

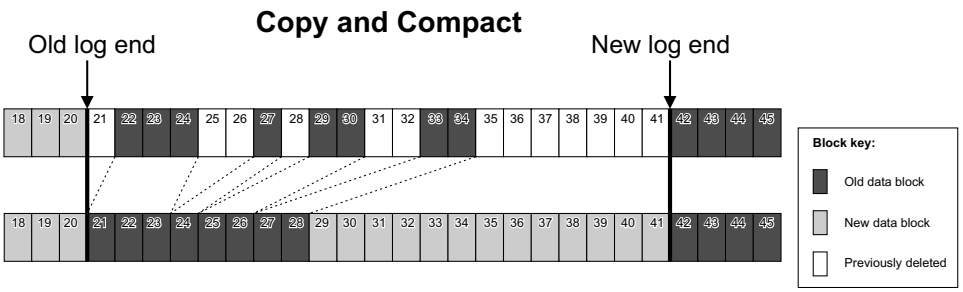

Figure 4.3: Copying technique. The log ends before and after the log wrap are shown.

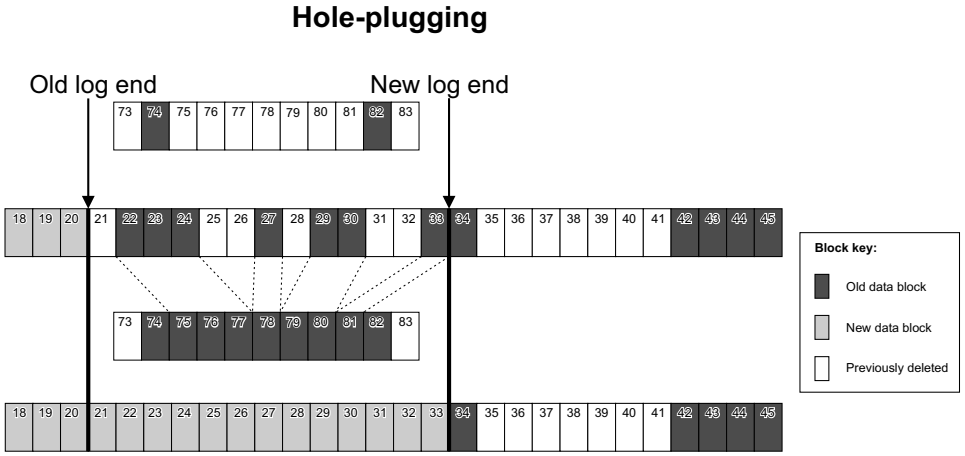

Figure 4.4: Hole-plugging technique. Parts of the log are shown before and after a log wrap.

free space become available. Then the threading technique can be used but it loses performance due to the empty space fragmentation problem. Newly written data gets fragmented.

Hole-plugging is a variation of copying that can still operate at a high disk capacity utilization. Fragmented log data is not copied back at the log end but is fitted into holes found in the fragmented log regions as shown in figure 4.4. Although the possible effects of data reconstruction can be abrogated by this approach, large free extents of free space can be generated for new log writes. This operation mode can be implemented in situations where a high disk capacity utilization makes the ideal copying approach not feasible any more due to the enormous time needed for it. The threading approach behaves differently to hole-plugging in that way that newly written data gets fragmented, whereas by hole-plugging existing data organizations are possibly broken into fragments. But careful algorithms can choose holes by which the data is still well organized. While existing log data is still copied around, the hole-plugging technique shows to be an alternative only in powered mode but the threading technique remains the better alternative in battery mode.

4.2.2.4 Combined Threading and Copying

The pure threading approach loses performance due to fragmentation issues. In contrast, the copying method controls the empty space fragmentation problem but loses performance by the copy operations. Therefore a combined approach is able

to compromise about the pros and cons of each method. After a log wrap fragmented live data should ideally be copied to the end of the log whereas unfragmented log regions should be skipped over. The combined threading and copying approach tries to approximate this behavior by dividing the log into extents of either fixed or variable size, which are generally referred to as segments in literature. Before a segment can be rewritten all live data has to be copied out of it. Therefore, segments form the unit of log threading. Once the current segment fills up, the nearest free segment is selected for servicing further write requests. After a log wrap segments have to be cleaned to satisfy new log writes. Which segments are selected for cleaning can be controlled by different cleaning policies. Each policy is determined by a different metric to estimate cleaning overheads. Then these overheads are minimized and appropriate segments are selected for cleaning. Unfortunately, a problem arises at a high disk utilization, especially for workloads with many random updates. Then to empty a segment many nearly full segments have to be cleaned. Hole-plugging addresses this problem by writing live blocks to the holes found in other segments instead of writing them to the end of the log. Hole-plugging can be combined with the other cleaning policies to form an adaptive one.

The hybrid approach as implemented in existing systems is not capable of data reorganization although this technique is essential for attaining better read performance, especially in battery mode operation. Therefore the segment cleaning policies have to be extended to work with the data reorganization scheme.

4.2.3 Crash Recovery

After the occurrence of a system crash the last performed file system operations can leave the file system in an inconsistent state. Then these operations have to be reviewed during the reboot phase to identify and then correct possible emerged inconsistencies. When file systems do not incorporate logging mechanisms the locations of changed data items cannot be easily determined after a crash. Then a sequential scan over all meta data structures is required to guarantee consistency of the file system. Fortunately, the log-structured approach easily identifies the locations which were affected by the last disk operations. They are always located at the end of the log. Therefore a quick crash recovery becomes possible by a two phase approach.

There checkpoints identify consistent states of the file system and roll-forward is used to recover data items that are written after the last checkpoint. Each check-

point should contain almost all recently performed file system modifications. Additionally, the consistency requirement must be provided for the log up to that point which requires to make many data structures persistent on disk. Because of that reason check-pointing can introduce significant overheads when applied frequently. For periodically applied checkpoints there is a tradeoff between check-pointing overheads and the time needed in the roll-forward phase. On the one hand long intervals between checkpoint creation are able to reduce overheads for check-pointing in the running system. On the other hand the non frequent creation of checkpoints implies a significant time increase of the recovery phase after a crash. Fortunately, recovering time can be limited by performing checkpoints only after a given amount of data has been written to the log. However, a low power file system should be able to provide both methods. The policy which selects and configures one method should be accessible from user-space. There it is possible to implement it by a daemon process. Then a more fine grained reaction on the different operating modes of the file system can easily be realized. Thus, it makes it possible to limit check-pointing in battery operation up to a certain degree to reduce energy overheads due to frequent check-pointing operations. Otherwise periodic checkpoints can be applied in active mode.

Chapter 5

Implementation

The concept presented in Chapter 4 has been implemented in the official Linux kernel source, version 2.4.21. The Linux kernel supports a variety of file systems via its internal virtual file system, also known as Virtual File System Switch or VFS. It is a kernel software layer which handles all system calls related to a standard Unix file system, thus providing a common interface to several kinds of file systems. Unfortunately, up to now none of the existing Linux file systems for block-oriented devices is purely log-structured. Only the Journaling Flash File System (jffs2) is an implementation of a log-structured Linux file system. However, it is designed to operate only with Flash memory technology as the underlying storage medium. Therefore, a completely new implementation was necessary. Although most of the provided switch functions from VFS were sufficient to interact with this implementation, the log-structured approach required a few changes to the Linux buffering mechanisms. In contrast, the directory handling mechanisms are adapted from the ext2 file system with a few minor modifications.

This chapter first describes the buffering schemes and policies provided by the Linux kernel. Then basic parts of the implementation are presented. Section 5.3 details the mechanisms which are needed for the management of the dynamically stored inode structures. Finally a new inode design is introduced and all changed and new implemented source files are listed.

5.1 Linux Disk Caches

In order to be able to understand some problems faced during the implementation process, a short outline of various Linux caching mechanisms is presented. Basi-

cally, in the Linux kernel framework performance optimizations for disk accesses are achieved by introducing four different caches:

Inode Cache. This cache is used for maintaining an in-memory representation of frequently used inodes. While almost all file operations need the corresponding inode object to execute, a speed-up in overall system performance is achieved.

Directory Cache (dcache). Since reading directory entries from disk and constructing the corresponding file system objects, called dentry objects, repeatedly causes disk accesses, the Linux kernel keeps them in memory depending on the available system memory. This speeds up the translation of a path name into the corresponding inode number. When a dentry is created the corresponding inode object is also loaded into system memory. Inodes which already have an in-memory representation and which are associated with an unused dentry structure are not discarded. Therefore, the associated inode of a dentry structure is always guaranteed to be also loaded into system memory.

Buffer Head Cache. The representation of a disk block in memory is described by `buffer_head` structures. Therefore the deletion and allocation of `buffer_heads` is an operation which takes place very often. The cache stores used and unused `buffer_heads` up to a certain amount to avoid this overhead.

Page Cache. The actual data of file disk blocks is stored in the page cache. The size of a page is dependent on the system architecture, but usually it is 4096 bytes large. Therefore it can store the associated data of a variable amount of `buffer_heads`. Each page in the cache corresponds to several blocks of a regular file or a block device file. All such blocks are logically continuous, thus representing an integral portion of a file. The page cache is indexed by file offsets and the memory address of the file operation pointer which is stored in the file's corresponding inode object.

Buffer Cache. As suggested by its name the buffer cache is a disk cache consisting of buffers. Indexing into the buffer cache is performed by device and logical block number. Usually, all disk blocks that contain meta data structures of a file system are cached in the buffer cache whereas all file contents are cached in the page cache. It is possible to have a representation of a logical disk block

both in the buffer and page caches, especially when the log-structured approach is implemented. Due to this fact the Linux kernel provides a function called `unmap_underlying_metadata` to make sure no disk request is issued by the update mechanism for that buffer after the return of that function.

5.2 The Log Approach

Although the implementation aims to operate without the caching of disk blocks, the buffering mechanisms are highly integrated in the Linux kernel. File systems save disk data into buffers which are located either in the buffer cache or page cache. Buffers are managed by three different lists for clean, locked and dirty buffers. When writing data to disk, buffers are simply marked dirty and are refiled to the dirty buffer list. Additionally, the buffer is given a future flush time which specifies how long the buffer may remain in the disk caches. Two kernel threads are actually responsible for issuing the write requests to the block device driver: the bdflush and kupdate daemon. The kupdate daemon is activated every five seconds to examine the head of the dirty buffer list and it will try to sync all buffers which need to be written to disk according to their flush time value. The bdflush daemon is activated when the kernel realizes that too many buffers are held in the disk caches. Then buffers are synced to disk. However, this approach takes away from the file system implementation the responsibility how and when a single block is written to disk. The file system is not able to order the write requests. Moreover, disk addresses have to be assigned to the block buffers when they are created. In contrast to update-in-place file systems, in a log-structured file system the disk address is assigned when blocks are written to disk instead of when they are written into the disk cache. Therefore, the Linux assumption that all blocks have disk addresses has to be addressed by the lazy assignment approach of log-structured file systems.

Scherlfs addresses this issue, which results from the log-structured behavior, both by providing mechanisms in the file system dependent code and by altering the code base of the buffer cache handling functions. Information about the file system is saved in the `super_block` structure which is located at the first logical disk block and is read in during the initialization phase. The function `scherlfs_get_block` is responsible for the translation of logical file blocks to disk blocks. First, existing blocks are passed by with their normal block numbers whereas new blocks are assigned a block number beyond the file system limit. Because it must be avoided that different file blocks are mapped to the same disk block a counter variable is

maintained in the `super_block` structure. Thereby it became possible to provide statically increased block numbers beyond the file system size limit when new blocks are going to be created. Second, all buffers are marked by setting in their state bits `BH_LOGFS`, which was added to the possible state values of the buffers. Therefore, buffers residing in a log-structured file system are easily identifiable. Third, there is a special list included, thus extending the least recently used lists for the different block buffers by maintaining a separate list for the dirty blocks of a log-structured file system. Therefore the `refile_buffer` function is modified in such a way that dirty blocks of a log-structured file system are appended to the logfs LRU list. Finally, the writing mechanism for blocks is based upon the corresponding inodes and not on the blocks themselves. All dirty blocks from one inode are written out together with the corresponding inode structure, thus forming the unit of disk writes. This makes it possible to re-map all dirty blocks for one inode when the inode itself is going to be written out to disk. The re-mapping of block buffers is done by requesting a log block which is accomplished by the function `log_getblk` and then the block number of the buffer block is simply altered to the new logical disk location. To make sure no disk I/Os are outstanding from within the buffer cache the function `unmap_underlying_metadata` has to be called on the buffer block. Another counter variable is increased when a disk block with a number beyond the file system limit is re-mapped. Therefore, it becomes possible to identify the state where all assigned block numbers beyond the file system limit are re-mapped and both counters can be reset. Finally, the update mechanisms are altered to ignore any blocks from log-structured file systems. The updates of their dirty blocks are performed at the granularity of write system calls.

5.3 Inode Map

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 must be implemented. This is done by a simple map lookup operation. 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 existent 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. There are three advantages to this approach. First it overcomes the limitation to be able to store 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 communicate to the file system via the standard file 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 an empty entry. Without applied optimizations a linear search has to be performed which should especially be avoided when the ifile is nearly full and therefore many ifile blocks have to be loaded and inspected for that purpose. 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. Thus, the search for an ifile block which has space for a new inode is reduced to the problem of finding the first zero bit in the bitmap. At initialization time the bitmap is simply filled with zeroes assuming that all blocks of the bitmap still have free places 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 ifile's inode is simply inserted into the superblock of the file system. The `read_inode` function is running into a bootstrap problem when reading in the ifile inode because a lookup into the inode map is required to determine the disk block of the requested inode structure which induces the need to read in the ifile inode. Therefore, the `read_inode` function must not perform a lookup into the inode map when the ifile inode itself is going to be read in, but it has to use the disk block number which is saved to the superblock.

5.4 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. In this scheme inodes store 15 block pointers. The first 12 components yield the corresponding data blocks directly to optimize the handling of very small files. The access to larger files is made

possible by the other indirect blocks. They point to blocks which contain either more file data blocks or indirect blocks. Therefore even second order or third order indirection is needed to enable the access to large files. To recognize the drawbacks of this scheme imagine a block size of 1 KB. With this mechanism the access of files up to a length of 12 KB executes with two disk requests: one to read the inode block and the other to read the requested data block. First indirection reveals the access to files with a maximal length of 268 KB but an additional disk request must be issued to read in the indirect block. Second and triple indirection even creates the need for additional disk requests. While disk accesses possibly cause seeks and rotational latency a new mapping scheme was implemented.

Whenever possible, dirty data blocks of a file are written to disk in large extents. Extents are triples that describe continuous 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. Thereby it is possible to save the file offset only implicitly. Then 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. Then 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 continuous 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. Then extents have to be split into two or three extents.

One major drawback of this scheme appears in an environment that has no disk caches. Then data blocks and inode blocks have to be written to disk at the granularity of system write calls. This is no problem when these calls are issued in large chunks of data. But applications mostly issue write calls with a buffer size that matches the system page size forcing the system to write out inode blocks followed by only a small amount of data blocks. This writing behavior of most Unix utilities causes an imperfect data layout on disk. Large continuous extents of data blocks are not possible any more because small extents are divided by inode

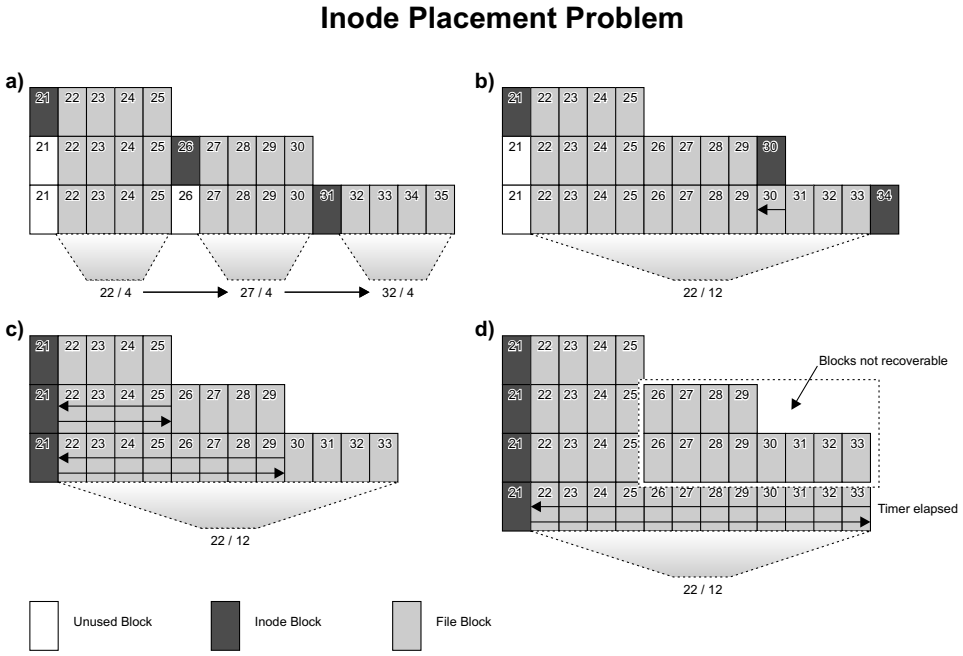

Figure 5.1: Different solutions to the inode placement problem. The resulting extent structures are shown below each diagram. The starting block number and the length of each extent is illustrated. Arrows within blocks indicate seek operations.

blocks on disk preventing to form larger ones. This scheme is implemented in scherlfs. It is illustrated in figure 5.1a. By remembering the inode number, which was involved in the last write operation, it becomes very easy to determine the situation when write calls are issued to the same file in order. Then there are three solutions to overcome the described inefficiencies.

First it would be possible to write the data blocks to the log before the corresponding inode structure (Figure 5.1b). When subsequent write calls to the same file are issued the last written inode structure can be overwritten with the data blocks which are selected to be written to disk. Then a new inode structure is appended. However, this approach makes the file system recovery code more complicated because the inode structures have to be linked together. A link which was integrated in the last inode structure would not point to an inode any more but to a file data block. Then the recovery code would have to check all following disk blocks until it would identify a correctly written inode structure or it would reach the log end. A second approach to the problem is to append all modified data blocks to the log as usual and then to write the new inode block over the disk loca-

tion of the last written one (Figure 5.1c). Unfortunately, the disk write location of the inode block will separate from the log write location, as the extent increases its length by the time, and so it becomes possible that the write operations of the inode block and the data blocks require two short disk seeks. This would introduce write inefficiencies which are not tolerable by the low power requirement of the file system design. Therefore the described method can be altered to delay the inode write operation a certain amount of time. Then the inode structure can easily be written over its previously used disk location. However, special care has to be taken for synchronized write requests where this scheme would not be applicable (Figure 5.1d), because it would not be guaranteed any more that previously successfully performed file writes are recoverable after a crash.

5.5 Modified and New Implemented Source Files

New Implemented Source Files

fs/scherlfs/balloc.c
fs/scherlfs/dir.c
fs/scherlfs/file.c
fs/scherlfs/ialloc.c
fs/scherlfs/inode.c
fs/scherlfs/namei.c
fs/scherlfs/super.c
fs/scherlfs/symlink.c
include/linux/scherlfs_fs.h
include/linux/scherlfs_fs_i.h
include/linux/scherlfs_fs_sb.h

Modified Source Files

fs/buffer.c

Chapter 6

Energy Measurements

The effects on power consumption of different file system layouts have not been investigated much up to now. This section compares the most popular Linux file systems with the newly implemented one (scherlfs) in regard to their energy requirements. First the measurement configuration is described. Then modifications to the Linux Kernel which were necessary to simulate a system with restricted use of the disk caches are explained and the used approach to simulate file system fragmentation is described. Finally the results of the energy measurements are presented.

6.1 Test Environment

6.1.1 Measurement Configuration

To validate energy savings of the new file system, a power measuring configuration is needed. One computer is used for monitoring purposes only. It is equipped with a four-channel analog-to-digital converter (ADC) developed at the University of Erlangen-Nürnberg. Thus, the measuring of the voltage drop at defined resistors in the power supply lines becomes possible with a resolution of 256 steps. The measuring device is capable of sampling at a rate of 20000 Hz. A maximum voltage drop of 50 mV is correctly converted. The software driver to read the data from the interface via the standard parallel port was written by Christian Winter [30].

Only one ADC channel is used with 10000 samples per second to measure the power dissipation of an IBM Travelstar mobile hard disk. A 100 mOhm resistor is used. Therefore the maximum voltage drop is equivalent to a dissipation of 5 W for

the Travelstar. Another computer is used as the target system. The IBM Travelstar is built into a Fujitsu Siemens Desktop PC.

All file system tests are carried out five times. Then the average of these test runs is calculated. Therefore variances between specific tests are smoothed out. The measurements of the different file systems represent the energy consumption on empty file systems. The measurements of scherlfs represent the energy consumption when no cleaner is present. These configurations yield the best operating conditions for the various file systems. During the tests hard disk power management was disabled. The tests are designed to contain no idle phases. Therefore, only the energy which is needed to perform the issued file system interactions is measured. This induces that the achieved energy savings are mainly caused by an increase in performance. It can be argued that a prolongation of idle periods is possible due to the saved time. This would increase energy efficiency in a real world situation even more when the power management of the hard disk is enabled.

6.1.2 Modifications to the Linux Kernel

While this thesis tries to evaluate the different file systems in an environment where not much memory is available, possible benefits of caching mechanisms have to be eliminated. Unfortunately the Linux kernel offers no general way to execute without its buffer and page caches and current file system implementations heavily rely on the Linux caching mechanisms. Therefore it was necessary to simulate an environment where mostly no cache is available by modifying the kernel sources directly. Two possibilities are able to reduce the mentioned effects.

First during the booting phase the Linux Kernel can be initialized so that it is allowed to use only a certain amount of the available memory. Depending on the running applications and on the used file systems a variable amount of system memory would be available for disk caching purposes in each case. Therefore for each different test and each file system the correct amount of kernel memory has to be determined experimentally to gain a fair evaluation basis. Even then it would not be guaranteed that no disk caches are used by the file systems during the measurements.

Another approach was developed in this thesis to gain a fair basis for comparison of the different file systems because most cache memory is used by the buffer and page caches. Therefore only these were restricted whereas the dentry and inode caches were not. At least little memory has to be reserved for them in a target system which has low system memory because enormous performance speed-

ups and energy savings are achievable by the use of these caches. Before reading blocks from disk a lookup into the buffer cache is always performed to check if the requested operation can be satisfied by the cache. For that reason the function `get_hash_table` will get called. It was modified to clear the `BH_Uptodate` flag of the requested buffer block when the block was found in the cache. However, this was only operable when the buffer was not in use which was tested by the `buffer_busy` macro, because all file systems rely on the fact that buffers which are marked to be used are not removed from the caches. Then the requested block will be read from the hard drive even if it was up-to-date beforehand. A similar approach is feasible for the page cache. Every access is performed in two steps. First the hash value of the page is calculated. Second the actual page is searched in the corresponding collision list by the function `__find_page_nolock`. This function was modified to simply change the page to the not up-to-date state by calling the `ClearPageUptodate` macro on it. Thereby it also has to be taken into account that used pages must be left in an unchanged state. As small read requests repeatedly call this function sequentially the described action is only performed when the requested page differs from the last requested one. Thus, small read requests are not needlessly penalized but the page cache is still reduced to the size of a single page. The results of these modifications ensure the execution of the `readpage` function for all read requests, which is defined by the `address_space_operations` object of the corresponding file system. It is responsible to read in all blocks of the specified page and it is implemented in all tested file systems by calling `block_read_full_page`, which creates or examines all block buffers within that page and issues read requests for all not-up-to-date block buffers. Again this behavior is modified to read in also the buffers in the up-to-date state which are not busy. Write requests that cover only partial pages are handled in Linux by reading in all block buffers of that page beforehand. Therefore the function `prepare_write` of the corresponding `address_space_operations` object of the file system is called when a page is going to be written to disk. This function is implemented in all tested file systems by the function `block_prepare_write`. Again this is modified to read in also the buffers in the up-to-date state which are not busy. In Linux modified block buffers are written to disk by first marking them dirty. Thereby they are refiled into the dirty buffers list. Then either the bdflush or kupdate kernel thread is responsible for the actual data transfer. To force a write out of the buffer at the same time when marking it dirty was not applicable because file systems like the fat32 would suffer from two seeks and rotational latencies when the FAT has to be

updated for each block. Thus, it was necessary to relax the no cache requirement by enforcing the write-out of all dirty buffers at the granularity of write system calls by issuing the functions `generic_osync_inode` and `sync_buffers` at the end of the execution of the function `generic_file_write` which is responsible for the write call.

By these modifications the no cache case is best simulated because the use of the disk caches is minimized and even none of the file system implementation is disadvantaged. The granularity of the updates at system calls is justified by the fact that additional memory requirements for the disk caches can be avoided by a careful implementation with no overheads.

6.1.3 Changed Source Files

These are the functions that are changed to simulate a system without disk caches in the Linux Kernel 2.4.21 to gain a fair evaluation basis among the different file systems:

get_hash_table	fs/buffer.c
block_read_full_page	fs/buffer.c
block_perpare_write	fs/buffer.c
__find_page_nolock	mm/filemap.c
generic_file_write	mm/filemap.c

6.2 Energy Consumption for Sequential Operations

The first comparative measurements examine the energy consumption between sequential read and write operations across a range of different file sizes. The data set consists of 50 megabytes of data, decomposed into the appropriate number of files for the file size being measured. 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. There are six phases to this test:

- **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 and their original size remains 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

The energy consumption of the various test phases for the different file systems is shown in Tables 6.1 - 6.10. Each table shows the energy consumption for sequential accesses of a different I/O size. It is maintainable to reserve a small amount of system memory in mobile computers for directory caching purposes when significant energy savings become possible. Therefore measurements for scherlfs are performed in two ways. The measurements for scherfs-d are different to scherfs in that way that all directory data is cached by the file system. This ensures that no seeks become necessary due to updates of directory data which is not cached.

In the create phase reiserfs shows 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 approach of the block groups shows to be inefficient. The ext3 file system behaves even worse because its recovering technique introduces additional overheads. Only as the file size reaches 1024 KB ext3 shows equal performance to ext2. At this point only 50 files are created. Therefore the introduced overheads by the journal updates are going to be negligible relative to the I/O transfer itself. Concerning the energy consumption the log approach of scherlfs proves to be optimal. Enormous energy savings become possible for all different file sizes. Relative to reiserfs energy savings of 66 % to 79.9 % are possible for the creation of small files with a size up to 32 KB. Although the relative savings decrease with increasing file sizes energy efficiency is still increased by 42 % for 512 KB files and by 37.4 % by 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 the directory updates.

The sequential read performance of the various file systems shows only minor differences. 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 in reiserfs's search

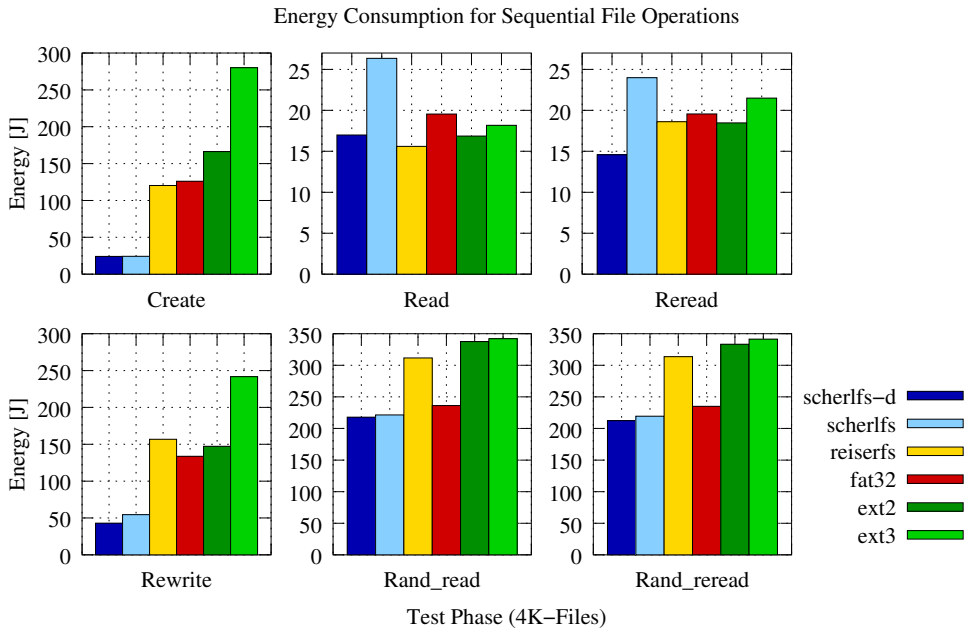

Figure 6.1: Energy Consumption of sequential file system accesses for different file operations with a file size of 4 KB

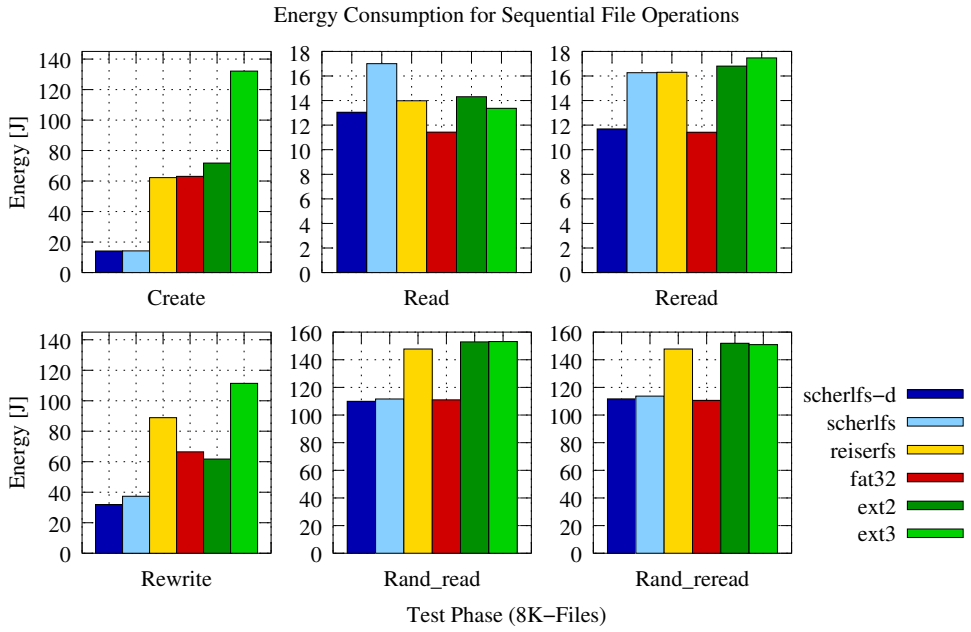

Figure 6.2: Energy Consumption of sequential file system accesses for different file operations with a file size of 8 KB

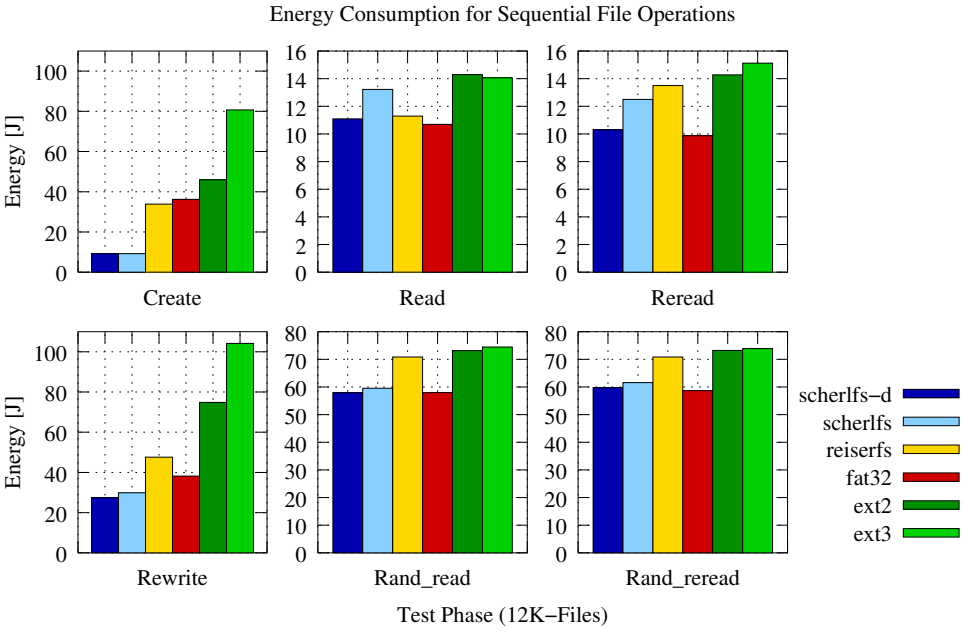

Figure 6.3: Energy Consumption of sequential file system accesses for different file operations with a file size of 12 KB

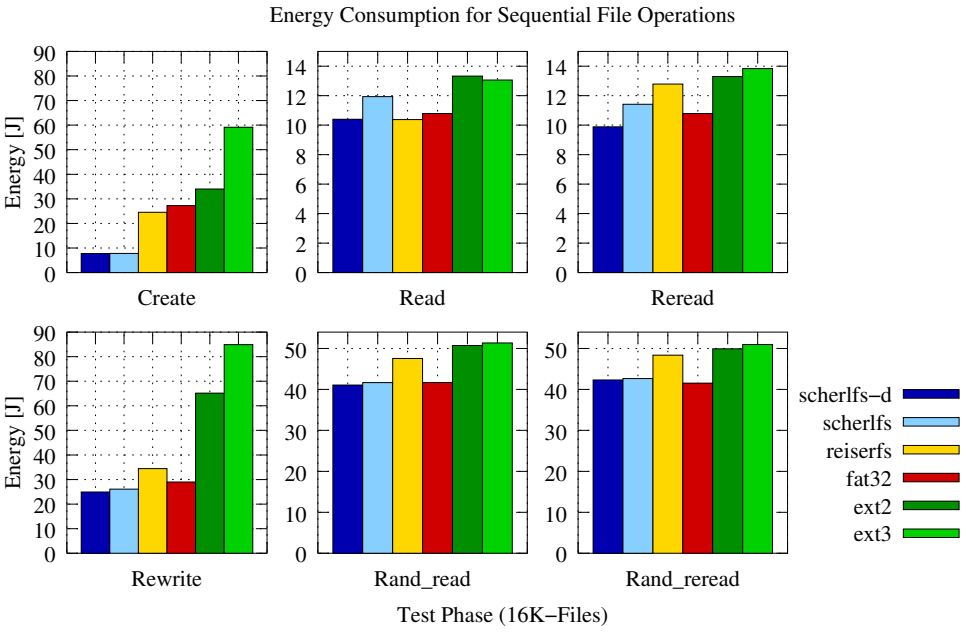

Figure 6.4: Energy Consumption of sequential file system accesses for different file operations with a file size of 16 KB

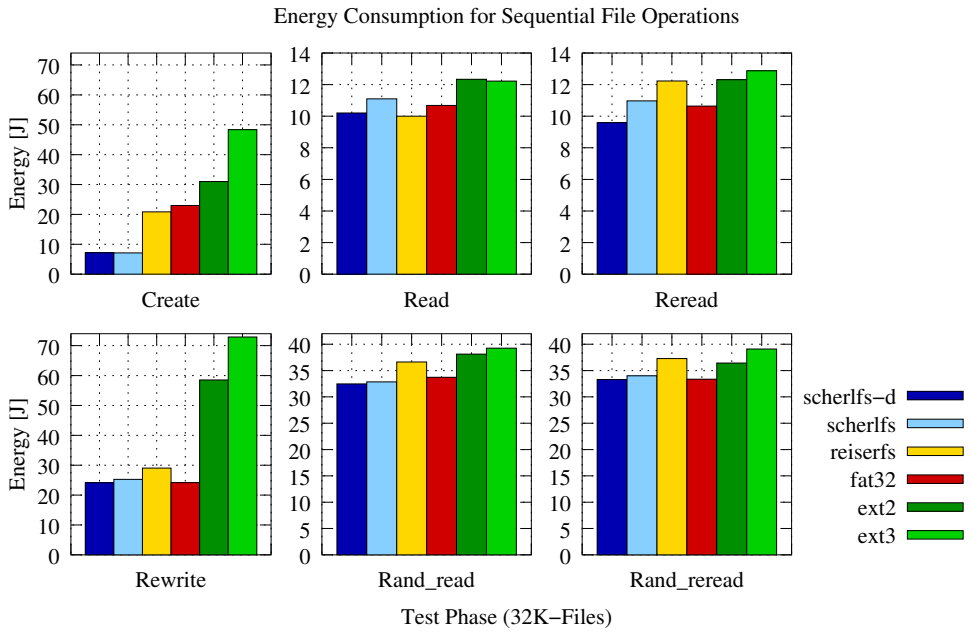

Figure 6.5: Energy Consumption of sequential file system accesses for different file operations with a file size of 32 KB

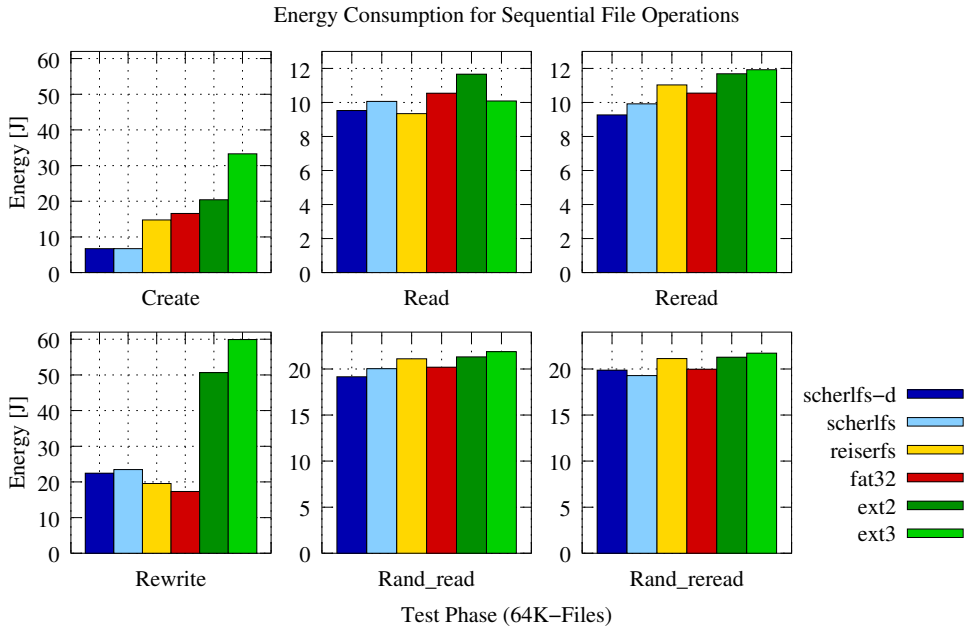

Figure 6.6: Energy Consumption of sequential file system accesses for different file operations with a file size of 64 KB

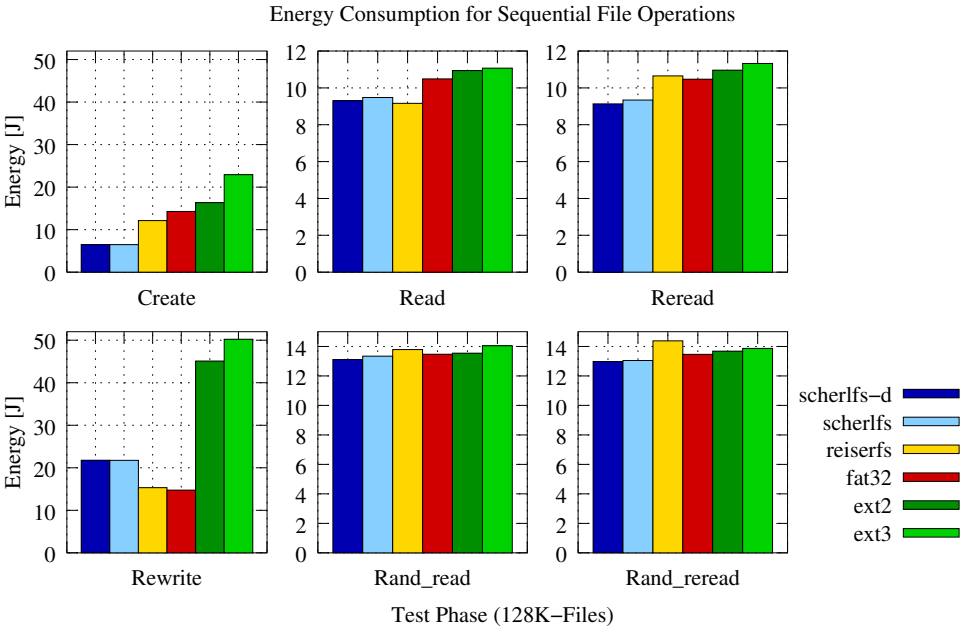

Figure 6.7: Energy Consumption of sequential file system accesses for different file operations with a file size of 128 KB

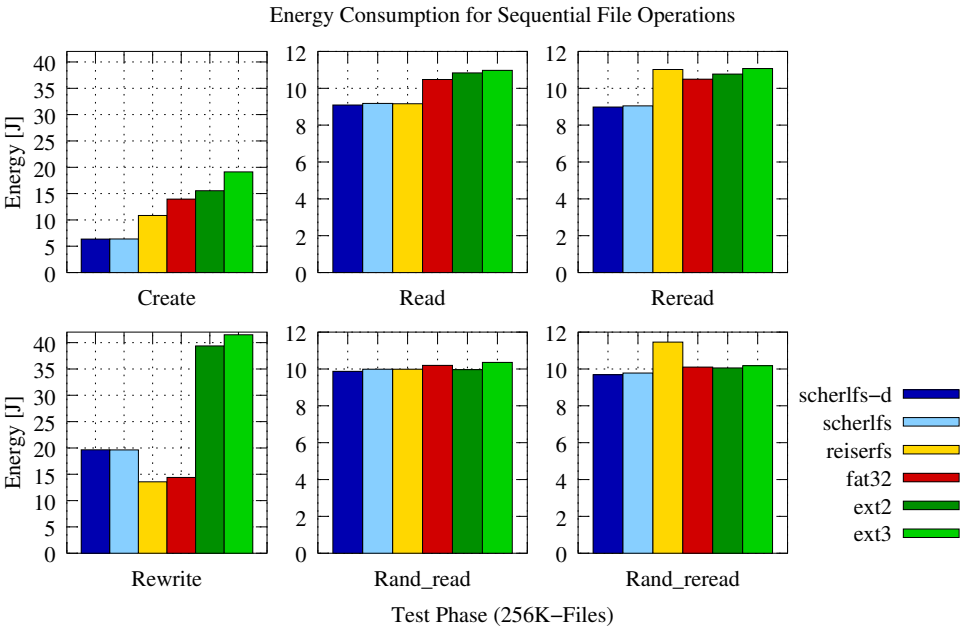

Figure 6.8: Energy Consumption of sequential file system accesses for different file operations with a file size of 256 KB

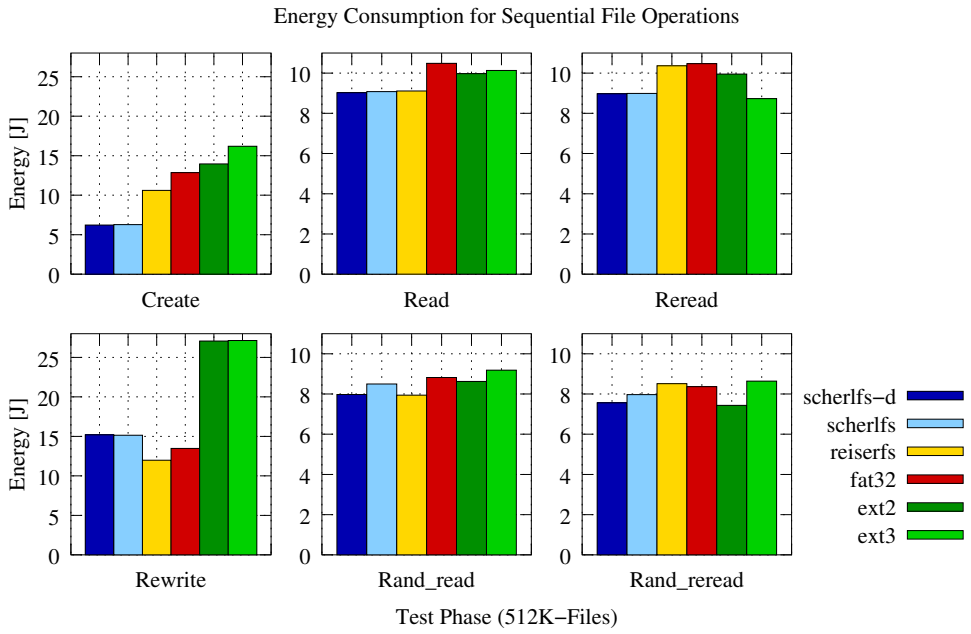

Figure 6.9: Energy Consumption of sequential file system accesses for different file operations with a file size of 512 KB

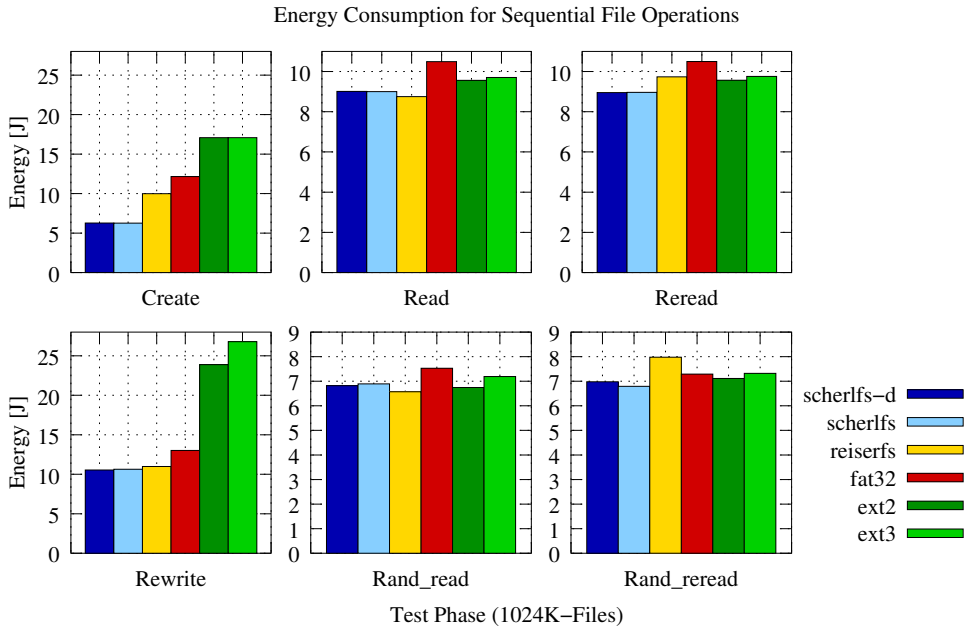

Figure 6.10: Energy Consumption of sequential file system accesses for different file operations with a file size of 1024 KB

tree directly. Therefore no additional seeks are required when the file object is found in the tree. The other file systems can only compete with it when the file size increases. Although scherlfs arranges all files sequentially in the log and the read order of the files is not varied, the directory lookups influence the energy consumption significantly. That's why it is not able to gain advantage of its file layout for small file sizes. For 4 KB and 8 KB file sizes its energy consumption shows to be very higher than that of reiserfs. Not until the file size increases over 32 KB does the influence of the directory lookups decrease because fewer files are going to be read in.

However, in the random read phase the file arrangement of scherlfs shows to be of good advantage for small file sizes because file meta data and file data are arranged sequentially on disk. Again, directory lookups influence the energy consumption strongly. While fat32 stores meta information for the files together in the directories and in its file allocation table, its energy requirements behave very similarly to scherlfs. But notice that this becomes possible only by the internal FAT cache of fat32, which is not eliminated by the modifications described in Section 6.1.2. For larger file sizes the file systems have only minor differences in energy consumption.

During the rewrite phase the energy consumption of scherlfs differs significantly. For small files up to a size of 32 KB the log approach shows to be of better performance relative to the other file systems. But not until file size reaches 1024 KB can scherlfs compete with other file system's energy efficiency. Scherlfs suffers from two restrictions. First directory lookups require disk seeks, then the inode lookup requires an additional disk seek and finally one additional seek has to be issued to reach the log end. Ext2 and ext3 have both very high energy requirements for all file sizes relative to the other tested ones which can be explained because of their frequent meta data updates.

After the rewrite phase the two read tests are performed on the updated file systems again. 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 overwrite them. Therefore, it becomes possible that the previous allocation scheme is 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. Therefore, the same file layout is created only at another disk location. The measurements show that the energy consumption decreases a little bit for scherlfs-d. As the files are

not deleted but only truncated, updates of directory information structures were not necessary. Therefore, only the data which corresponds to the file updates is appended to the log. Now no directory data is saved in between the files by the performed log appends. Therefore, 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 achieved energy savings. Even without an enabled directory cache scherlfs was able to gain advantage of the new file system layout, but more little energy savings were achieved.

The measurements showed that scherlfs offers energy improvements especially for the creation of new files. Its energy efficiency is often better or at least comparable to the other file systems in the other test phases. However, for some test phases it cannot reach the energy efficiency of other file systems. That's mostly caused by directory lookup operations which avoid gaining the advantages of the created file layout. When directory information is cached file system performance for scherlfs could be improved significantly.

6.3 Energy Consumption for Random Updates

The second comparative analysis examines the energy consumption of the different file systems when small random I/Os are issued to one large file. Therefore, a file of 100 megabytes length is created within this test. There are five phases to this test:

- **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 amount 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 amount 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 6.11 shows the energy consumption of the described test phases for the different file systems. As the target file is created on a new file system the test file

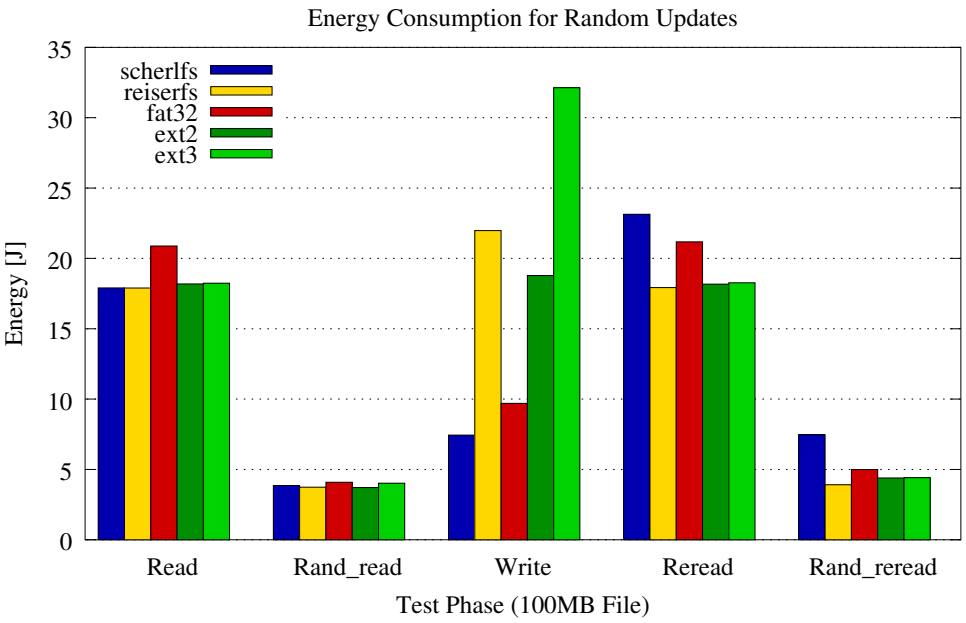

Figure 6.11: Energy Consumption for large files which are randomly updated by 8 KB I/O operations.

was able to be created in an optimized manner by each file system. Therefore, all file systems show the same energy measurements during the first sequential and random read phases. However, fat32 consumed nearly three Joules more than the best file systems in the read phase, due to the fact that the fat32 file system must perform a lookup in its file allocation table every time when a new block of the file is going to be loaded into memory. This requires two additional disk seeks when the corresponding FAT block was not already 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. Therefore, the additional seeks are only required when a FAT lookup affects a block which is not loaded into the cache at that time. Therefore, 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 write with lowest energy requirements because the updated file data must only be appended 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 in 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 performed 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 split within the log significantly.

6.4 Testing a Digital Camera Appliance

The next step was to examine whether the new concept is able to save energy in a real world application at all. Digital cameras take pictures and write them to disk immediately. Thus, this workload is characterized by mostly writing out mid sized files. By this test it was possible to compare the energy consumption of file systems when writing out new files. As digital cameras are embedded devices their behavior has to be simulated at the target workstation computer for this test. This is done by a perl program. It reads in 156 photo files, which are together 145.8 MB large, from a three Mega Pixel Canon Powershot S30 digital camera. The average file size was 957 KB. Then all files are written out to disk subsequently.

Figure 6.12 shows the results of the performed energy measurements for the

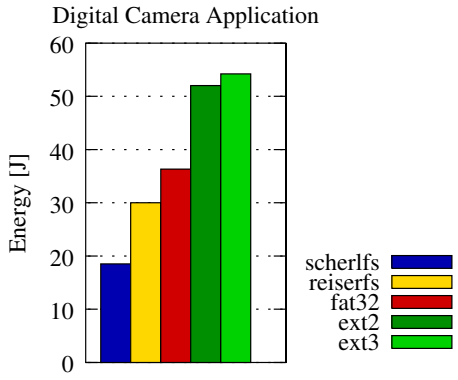

Figure 6.12: Energy Consumption for a digital camera application

different file systems. They attest scherlfs an optimized energy use. In comparison to the energy consumed by reiserfs relative savings of 38.25 % are possible by the new file system design. Scherlfs is able to write out the files with minimal overheads 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 worst energy consumption. This is caused by 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 test shows that significant energy savings can be reached when the overheads of traditional disk managers are avoided.

6.5 Testing an Mp3 Player

The workload of a mp3player appliance is ideally suited for examining the power consumption when reading large files. In this test 35 mp3-files are played. Overall 138.4 MB of data is examined by the player. The average file size was 3.95 MB.

Figure 6.13 shows the results of the energy measurements. It can be seen that the different file systems perform equally well in reading the large mp3-files. The slightly worse file system performance of the ext2 and ext3 files can be explained by the file data organization into block groups. Therefore, the file data is not organized completely sequentially. As all mp3-files are played in their creation order this arrangement penalizes the two file system. However, the test shows that not much energy savings are possible in the case of very large files.

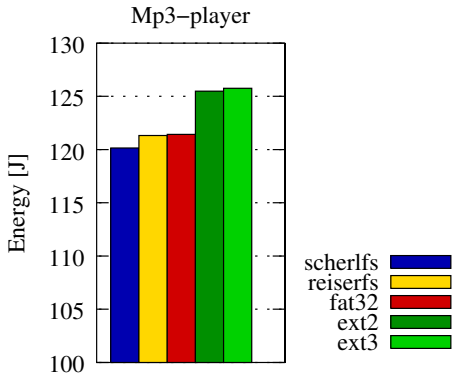

Figure 6.13: Energy Consumption for a mp3-player application

6.6 Effects of Fragmentation on Energy Consumption

All file systems rely on the allocation of contiguous disk blocks to achieve high levels of performance. By the design described throughout this thesis the fragmentation problem for scherlfs is reduce-able by data optimization and defragmentation techniques which can be applied in powered mode. As mentioned before the performance of the other file systems is susceptible to fragmentation, nevertheless. Now the effects of free space fragmentation are examined for different levels of fragmentation. Here the digital camera appliance as an example for a write dominated workload and the mp3-player application for a read dominated workload are chosen for comparison. Therefore, a method has to be found to create the same amount of fragmentation at each file system to be able to compare the resulting energy dissipations. All measurements are performed within a newly created file system and one that is affected from 20 %, 40 %, 60 % and 80 % fragmentation. To simulate the fragmentation for each file system an image was created for each fragmentation level. Then this image was copied to the partition in the test preparation phase. For the creation of the images a perl program was written. In the first step the program wrote 8 KB files to the file system until no space was left on it any more. Then files were pseudo randomly chosen for deletion until the file system capacity utilization would reach a specified percentage.

Figure 6.14 shows the effects of fragmentation on the digital camera application which is presented in Section 6.4. The measurements show that the different file systems lose energy efficiency at higher fragmentation levels when writing to disk new files. All tested file systems have to invest 15 to 23.9 % more energy to fulfill the same job for a fragmentation level of 20 %. In the 40 % fragmented scenario the energy inefficiency increases up to 50 %. At an 80 % disk utilization the energy use increases more than 200 % relative to each file systems ideal operating conditions. The new proposed file system design already shows an enormous increase in energy efficiency in the no fragmented case. If the device usage allows to execute its cleaning strategies in powered mode, its energy requirements will not increase. Therefore, significant energy savings become possible. Figure 6.15 shows the energy measurements of the mp3-player application of section 6.5 under different levels of fragmentation. In contrast to the enormous savings, which are achievable when writing new files to disk, fragmentation shows to be of minor importance to read performance. Only small benefits around 5 % are attainable at lower fragmentation levels. When disk reads are often performed for the same

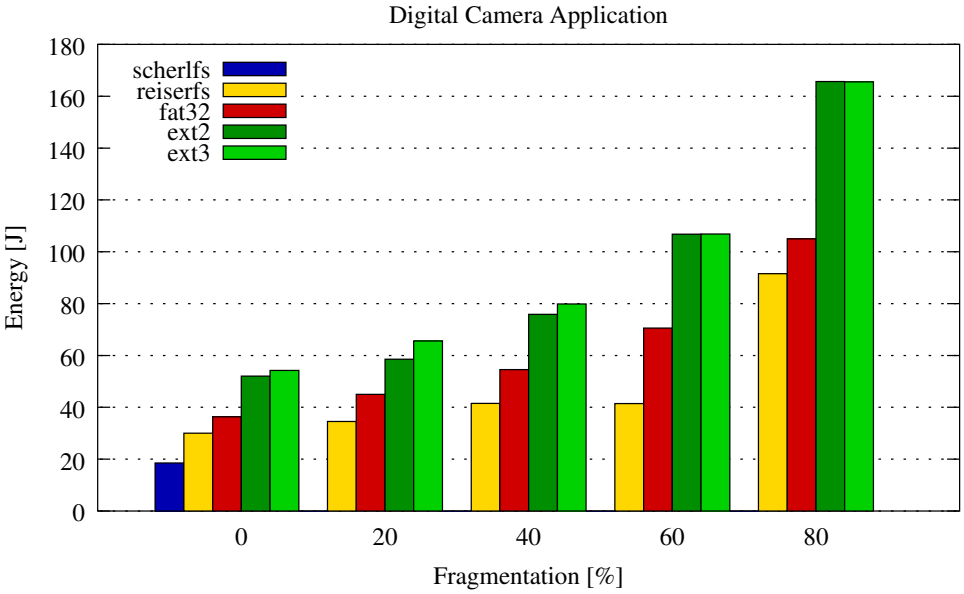

Figure 6.14: Effects of Fragmentation on a digital camera application

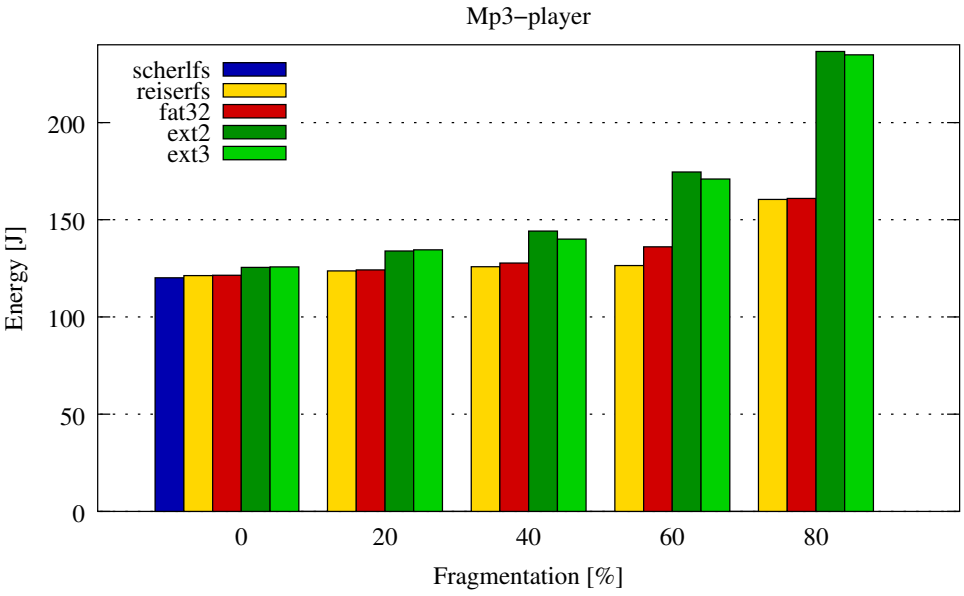

Figure 6.15: Effects of Fragmentation on an mp3-player application

files the energy savings will sum up to a remarkable amount, nevertheless. At a very high disk utilization file system performance degrades enormously. Then the reduction of fragmentation can increase energy efficiency significantly.

Chapter 7

Future Work

7.1 Directory Cache

The energy efficiency of the new file system design is very sensitive to directory lookup and directory update operations in most situations. It is possible to reserve some memory for a special directory cache. This behavior is only simulated up to now. A clean and careful implementation would be an important objective of future work.

7.2 Improvement of Random Update Performance

The measurements described in Section 6.3 attest scherlfs bad energy efficiency for random file updates. Therefore, ways to improve the situation have to be found to extend the use of application of the file system design. The file system can be altered to observe the file update characteristic and change from the append only semantic of the log-approach to an update-in-place semantic when file changes are applied randomly to an existing file.

7.3 Overhead of Cleaning Strategies

In section 4.2.2 different strategies for free space generation are discussed. Up to now none of these strategies is implemented. In the future the overheads of each strategy can be examined for the different modes of operation.

7.4 Effects of Data Reorganization Techniques

It is possible to detect fragmented files by the new inode design easily because the amount of extents in relation to the file size would be a good indicator for fragmentation. Therefore these files should be preferably chosen to be reorganized. This means to rewrite them to the end of the log in a single extend. Furthermore, past accesses to files can be used to predict the access order of files. When these accesses are logged data reorganization can be extended to match file layout to these access characteristics.

7.5 Undo operations

As the log semantic does not overwrite file contents when they are updated it would be possible to allow users to undo last file operations up to a certain limit. Especially for mobile computers, which are sometimes used without much care, undo possibilities would be very helpful to the users.

Chapter 8

Conclusions

This work examines the energy efficiency of different file system layouts. This is done in an environment where the use of disk block caching techniques is not possible because the available system memory is used by the applications and the operating system. This is often the case in mobile devices and mobile computers.

Concerning the file system layout the areas where energy savings become possible are explored. It turned out that the avoidance of disk seeks and rotational latencies is the most promising candidate for improving energy efficiency. Guidelines for a low power file system design are developed. The most important points that are suggested for such a design are the reduction of meta data updates, a sequential arrangement of meta data and data blocks and the enabling of a sequential reading and writing behavior. For that reason file system reorganization techniques and an adaptive energy use, depending on the kind of the power supply, must be incorporated into the file system. Finally the possibility for fast crash recovery must also be given while mobile systems often fail because of power reasons and shutdown actions by users. The proposed file system design is implemented in the Linux kernel 2.4.21. A log-structured approach was chosen to fulfill the proposed goals. The problem of free space generation and cleaning which becomes necessary by the log-structured way are thoroughly discussed.

Energy measurements both for synthetic tests and real world applications are performed for various Linux file systems and the newly implemented one. The measurements attest the new file system design enormous energy saving potential when new files are written to disk. The energy efficiency of workloads which are characterized by mostly sequential file operations can also be improved by the new file system design. The measurements demonstrate that a specially designed directory cache is able to improve energy efficiency of the log-structured design

even more. For some sequential workloads energy savings only become possible with enabled directory caching. In the case of a workload that randomly updates small file portions the new file system design shows to be inefficient in comparison to the other update-in-place file systems. In conclusion the effects of fragmentation on the energy efficiency are analyzed. With the described data reorganization techniques that can be performed when the device is attached to an external power supply the new file system design is able to improve energy efficiency even more.

Bibliography

[1] Compaq, Intel, Microsoft, Phoenix, and Toshiba. *Advanced Configuration and Power Interface Specification 2.0b*, October 2002.

[2] Christian Czezatke and M. Anton Ertl. LinLogFS — a log-structured filesystem for Linux. In *Freenix Track of Usenix Annual Technical Conference*, pages 77–88, 2000.

[3] F. Douglis, P. Krishnan, and B. Bershad. Adaptive disk spindown policies for mobile computers. In *Proceedings of the Second USENIX Symposium on Mobile and Location Independent Computing*, Apr 1995.

[4] Fred Douglis, P. Krishnan, and Brian Marsh. Thwarting the power-hungry disk. In *USENIX Winter*, pages 292–306, 1994.

[5] C. Ellis. The case for higher level power management. In *Proceedings of the Seventh Workshop on Hot Topic in Operating Systems HotOS 1999*, March 1999.

[6] Jason Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. Technical report, School of Computer Science, Carnegie Mellon University, 1999.

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

[8] Taliver Heath, Eduardo Pinheiro, and Ricardo Bianchini. Application-supported device management for energy and performance. In *Proceedings of Workshop on Power-Aware Computer Systems PACS'02*, February 2002.

[9] Taliver Heath, Eduardo Pinheiro, Jerry Hom, Ulrich Kremer, and Ricardo Bianchini. Application transformations for energy and performance-aware

device management. In *Proceedings of the Eleventh International Conference on Parallel Architectures and Compilation Techniques PACT'02*, September 2002.

[10] David P. Helmbold, Darrell D. E. Long, and Bruce Sherrod. A dynamic disk spin-down technique for mobile computing. In *Mobile Computing and Networking*, pages 130–142, 1996.

[11] P. Krishnan, Philip Lon, and Jeffrey Scott Vitter. Adaptive disk spindown via optimal rent-to-buy in probabilistic environments. *Algorithmica*, 23(1):31–56, 1999.

[12] Kester Li, Roger Kumpf, Paul Horton, and Thomas Anderson. A quantitative analysis of disk drive power management in portable computers. In *Proceedings of the USENIX Winter 1994 Technical Conference*, January 1994.

[13] Jacob R. Lorch and Alan Jay Smith. Software strategies for portable computer energy management. Technical Report CSD-97-949, 13, 1997.

[14] Y. Lu and G. De Micheli. Adaptive hard disk power management on personal computers. *IEEE Great Lakes Symposium on VLSI*, pages 50–53, 1999.

[15] Yung-Hsiang Lu, Luca Benini, and Giovanni De Micheli. Requester-aware power reduction. In *International Symposium on System Synthesis*, pages 18–23. Stanford University, September 2000.

[16] Yung-Hsiang Lu, Luca Benini, and Giovanni De Micheli. Power-aware operating systems for interactive systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 10(2), April 2002.

[17] Yung-Hsiang Lu and Giovanni De Micheli. Comparing system-level power management policies. *IEEE Design & Test of Computers*, 18(2):10–19, March 2001.

[18] Microsoft. OnNow Power Management.

[19] 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*, pages 238–251. ACM Press, 1997.

[20] Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile application-aware adaptation for mobility. In *Sixteen ACM Symposium on Operating Systems Principles*, pages 276–287, Saint Malo, France, 1997.

[21] A. Papathanasiou and M. Scott. Increasing disk burstiness for energy efficiency. Technical Report 792, Department of Computer Science, University of Rochester, November 2002.

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

[23] J. Schindler, J. Griffin, C. Lumb, and G. Ganger. Track-aligned extents: matching access patterns to disk drive characteristics. Conference on File and Storage Technologies, 2002.

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

[25] Margo I. Seltzer, Keith A. Smith, Hari Balakrishnan, Jacqueline Chang, Sara McMains, and Venkata N. Padmanabhan. File system logging versus clustering: A performance comparison. In *USENIX Winter*, pages 249–264, 1995.

[26] John Zedlewski Sumeet. Modeling hard-disk power consumption. Department of Computer Science, Princeton University.

[27] Hitachi Global Storage Technologies. Hitachi familiy of microdrives datasheet. http://www.hgst.com/hdd/micro/datasheet.pdf.

[28] Amin Vahdat, Alvin Lebeck, and Carla Ellis. Every joule is precious: A case for revisiting operating system design for energy efficiency. In *Proceedings of the 9th ACM SIGOPS European Workshop*, September 2000.

[29] Andreas Weissel, Bjoern Beutel, and Frank Bellosa. Cooperative I/O: A novel I/O semantics for energy-aware applications. In *Proceedings of the Fifth Symposium on Operating System Design and Implementation OSDI'2002*, December 2002.

[30] Christian Winter. Measuring power consumption with linux. Pre-Master's Thesis, Department of Computer Science, University of Erlangen/Nürnberg, 2002.

[31] H. Zeng, C. Ellis, A. Lebeck, and A. Vahdat. Currentcy: Unifying policies for resource management. In *Proceedings of the USENIX 2003 Annual Technical Conference*, June 2003.

[32] Heng Zeng, Xiaobo Fan, Carla Ellis, Alvin Lebeck, and Amin Vahdat. Ecosystem: Managing energy as a first class operating system resource. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems ASPLOS'02*, October 2002.

Design und Implementierung eines energiesparenden Dateisystems

In den letzten Jahren kamen eingebettete Systeme und mobile Geräte immer mehr in Mode. Mobile Computer sind in höchstem Maße von ihren Batterien abhängig, da sie, wie ihr Name schon sagt, in einem mobilen Umfeld eingesetzt werden. Sie müssen deshalb über einen längeren Zeitraum ohne externe Stromzufuhr betrieben werden. Deshalb ist die Reduzierung des Energieverbrauchs solcher Geräte zu einer der wichtigsten Aufgaben des Designs mobiler Systeme geworden. Zugleich erweiterte die Miniaturisierung von Massenspeichermedien und deren Integration in mobile Geräte die Möglichkeiten mobiler Anwendungen und somit deren Nützlichkeit für den Endverbraucher. Somit zählt die Festplatte als Massenspeichermedium in heutigen mobilen Rechnern mit zu einem der größten Stromverbraucher und eröffnet deshalb ein großes Potential für eine Steigerung der Energieeffizienz mobiler Geräte.

Fast alle Komponenten mobiler Geräte können in so genannten Niedrigenergiemodi betrieben werden, wenn sie gerade nicht benutzt werden. Auch mobile Festplatten stellen verschiedene Betriebsmodi zur Verfügung. Allerdings kostet der Übergang zwischen den verschiedenen Modi sowohl Zeit als auch Energie. Deshalb muss die Festplatte eine genügend lange Zeit in den Energiesparmodi betrieben werden, damit überhaupt eine Energieeinsparung und nicht etwa sogar eine Energieverschwendung erreicht wird. Bisherige Verfahren zur Energieeinsparung versuchen die verschiedenen Betriebsmodi optimal auszunutzen.

In dieser Arbeit wird ein anderer dazu komplementärer Ansatz untersucht. Bei Lese- und Schreibvorgängen muss die Festplatte ihren Schreib/Lese-Kopf neu positionieren und abwarten, bis die entsprechende Stelle der Magnetplatte darunter rotiert. Diese beiden Vorgänge haben einen nicht unwesentlichen Anteil am Energieverbrauch der Festplatte. Normalerweise können viele dieser Latenzen durch das Cachen einmal gelesener Diskblöcke absorbiert werden. Allerdings muss

für eine effektive Anwendung der Cachingtechnik ein nicht unwesentlich großer Hauptspeicherbereich zur Verfügung stehen. Gerade mobile Systeme sind so konzipiert, dass der zur Verfügung stehende Hauptspeicher fast vollständig von den laufenden Anwendungen und dem Betriebssystemkern benutzt wird. Eine Erweiterung des Hauptspeichers kann durch die hohen Kosten und durch den Energie- und Platzverbrauch der Speichermodule in den meisten Fällen nicht vorgenommen werden. Durch ein optimales Dateisystemlayout können aber viele dieser Latenzen vermieden und dadurch Energieeinsparungen erreicht werden.

Mehrere Aspekte, die ein energiesparendes Dateisystemdesign enthalten sollte, sind untersucht worden. Es stellte sich heraus, dass ein log-strukturierter Ansatz am besten geeignet erscheint um die gemachten Vorschläge zu verwirklichen. Probleme, die eine solche Herangehensweise mit sich bringt, sind detailliert besprochen worden. Darauf aufbauend wurde ein neues Dateisystem implementiert und vorgestellt. Um Messungen unter Linux ohne beträchtliche Einflüsse der Diskcaches durchzuführen, wurde der Kern sorgfältig modifiziert, aber immer im Hinblick darauf, noch eine faire Vergleichsgrundlage zwischen den einzelnen Dateisystemen zu gewährleisten. Durchgeführte Messungen attestieren dem neu implementierten Dateisystem in einigen Bereichen signifikante Energieeinsparungen. Vor allem für das Schreiben neuer Dateien kann mit dem vorgeschlagenen Ansatz eine enorme Energieeinsparung erreicht werden. Auch sequentielle Dateisystemoperationen eröffnen Einsparpotential insbesondere für kleine Dateigrößen. Allerdings gibt es auch Fälle, wo eine Energieeinsparung nicht erreicht wird, ja sogar eine Energieverschwendung im Vergleich zu den existierenden update-in-place Dateisystemen auftritt. Das ist insbesondere bei wahlfreiem Zugriff auf große Dateien der Fall. Auch stellte sich heraus, dass der vorgestellte Ansatz höchst sensitiv auf Verzeichniszugriffe und -modifikationen reagiert. Erst durch das Puffern dieser Strukturen können die erhofften Energieeinsparungen dieses Ansatzes vollends ausgeschöpft werden.