Universität Karlsruhe (TH)
Research University • Founded 1825

System Architecture Group
Department of Computer Science

Improving Energy Efficiency Through File Compression

Clemens Koller

Study Thesis

Adivsors:	Prof. Dr.-Ing. Frank Bellosa
	Dipl.-Inf. Andreas Merkel

June 3, 2008

Abstract

Because of the rising energy costs and the increased energy consumption of computer systems it is becoming more and more important to increase energy efficiency. To achieve this modern operating systems use several power management strategies which mostly focus on reducing the energy consumption of the CPU. Other components like hard disk drives are often neglected and are at best just turned off if not needed.

To also increase the energy efficiency of systems where the disk drives are not idle long enough to be turned off, we propose to utilize compression to reduce disk activity. Due to compression more data can be stored in caches in main memory and thus the disk has to be accessed less often, and reading or writing the compressed data from disk takes less time and therefor less energy.

To evaluate if the energy saved by the disk drive outweighs the additional energy needed by the CPU for compression and decompression we implement a block device driver that transparently compresses all data written to disk. The results of our measurements show that on desktop systems, where the CPU consumes significantely more energy than the disk drive, no energy can be saved using this approach. On embedded systems and mobile devices though, where the CPU and the disk energy consumption are similar, it is possible to improve energy efficiency by upto 50% while at the same time also increasing I/O performance.

Contents

Chapter 1

Introduction

Along with the increase of computing power over the last years also came an even faster increase of power consumption of computer systems. Because of the rampant energy costs and the only slowly advancing battery technologies this has become a growing problem and the importance of high energy efficiency is rising.

To reduce the energy costs and cooling needs of desktop and server systems, and to extend the battery life of mobile devices, modern operating systems use several power management strategies. Most of those strategies concentrate on reducing the energy consumed by the CPU though while ignoring most other parts of the system. But thanks to the improvements in processor technologies the CPU is no longer the all-dominant energy consumer in a computer system anymore which means the focus of power management has to shift to other components.

Hard disk drives are one of those components, especially considering the need to store more and more data. They are used as the primary data storage in consumer PCs and because of their low cost and high performance are also starting to replace tape devices as digital storage for archival purposes, and in the form of microdrives they are even available for small portable devices. This results in the need of new ways to increase the energy efficiency of disk drives.

1.1 Motivation

Besides the classical field of application of data compression—better utilization of limited storage space and the acceleration of data transmission—it can also be used to improve energy efficiency: memory modules no longer needed due to compression can be deactivated and especially in wireless communication a significant amount of energy can be conserved when less data has to be transmitted.

Following this line of thought, it should also be possible to increase the energy efficiency of disk drives by using compression. Compared to the main memory compression mentioned above the data stored on disk does not consume any energy though, only reading and writing it does. Therefore disk accesses need to be avoided or at least their length needs to be shortened to save energy. This can by achieved by compressing the data before writing it to disk and decompressing it after reading. And as long as the energy saved by the disk outweighs the additional energy consumed by the CPU for compression the overall system energy efficiency can be improved.

1.2 Proposed Solution

Based on this idea our solution is to put an additional layer between the file system code and the disk drive which transparently compresses all data. This way we are independent from the file system and the actual disk drive used and thus this approach is applicable for a wide range of systems.

To further reduce the number of disk accesses a compressed cache in main memory is used. Compared to the regular uncompressed disk cache this allows us to store more data in memory so less has to be read from and written to disk. Especially on embedded systems with little memory this can greatly reduce the energy consumed by the disk.

1.3 Structure of the Thesis

The thesis is structured as followed:

- *Chapter 2* provides background information about how energy consumption can be reduced when using disk drives and lists related work where compression has been used to improve energy efficiency.
- *Chapter 3* explains the design and internal details of our approach to conserve energy by reducing disk activity.
- *Chapter 4* covers the implementation details of our solution.
- *Chapter 5* describes the two test systems used and discusses the obtained experimental results.
- *Chapter 6* is a summary of the thesis and also includes directions for further work.

Chapter 2

Background and Related Work

The first part of this chapter provides background about *hard disk drives* (HDDs) and their power management, whereas the second part gives an overview of existing approaches to reduce energy consumption with compression.

2.1 Background

To improve the energy efficiency of disk drives it is important to understand how a HDD works and which parts of it consume the most energy.

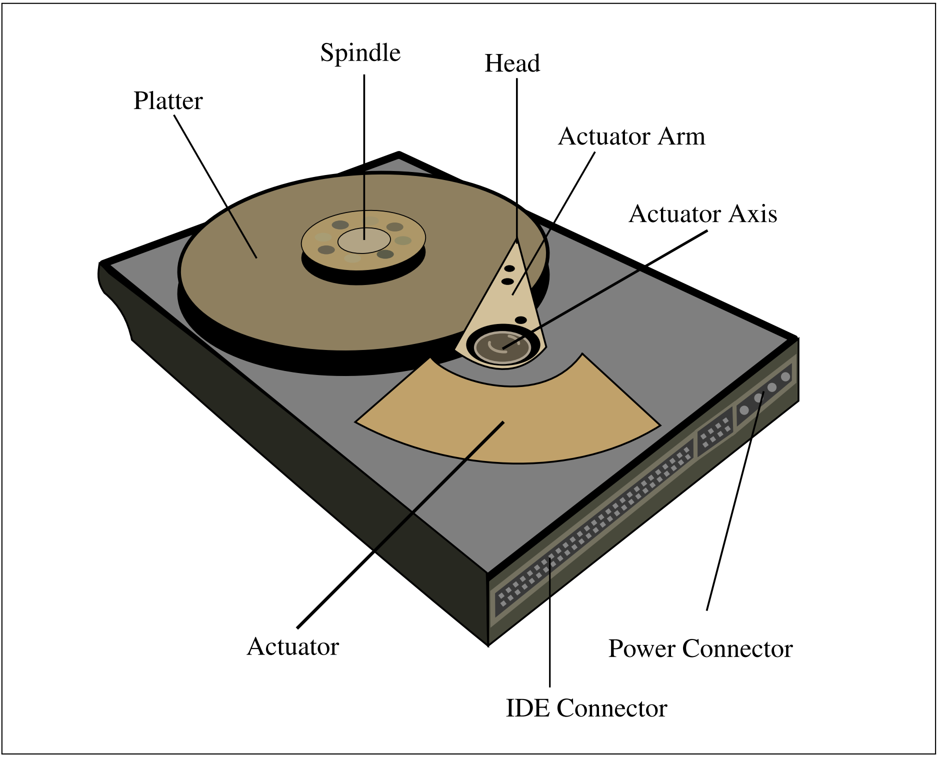

Figure 2.1: Schematic view of a hard disk drive. (Source: [Wik08]) .

2.1.1 Hard Disk Drives

Figure 2.1 shows a schematic view of the components of a HDD. The Platter is a circular disk made of ferromagnetic material which is rotating at very high speeds, from 5400 *revolutions per minute* (rpm) in notebook HDDs up to 15000 rpm in high-end server drives. Information is recorded to it by magnetizing small areas, representing a binary 1 or 0. This is done by the read-and-write head which hovers very closely (only tens of nanometers in modern drives) over the platter. To read or write data this head first has to be brought into the right position over the platter by the actuator, then has to wait till the right area of the spinning platter arrives under it.

Thus there are four primary energy consumers in a HDD: the spindle motor, the actuator, the read-and-write head and the electronics operating the disk's interface and controlling the other components.

2.1.2 Hard Disk Drive Power Management

To reduce energy consumption a modern disk drive is able to selectively turn off unused parts by supporting several power modes. Table 2.1 shows the power modes of an IBM Travelstar and their corresponding power usage. Also shown is the drawback of the power modes: when they are used the latency to access data can increase significantly.

Mode	Properties	Power	Latency
Active	reading/writing	2.1-4.7 W	-
Performance Idle		1.85 W	-
Active Idle	actuator turned off	0.85 W	20 ms
Low Power Idle	head parked and off	0.65 W	300 ms
Standby	spindle motor off	0.25 W	1.0-9.5 s
Sleep	electronics off	0.1 W	3.0-9.5 s

Table 2.1: Power modes of an IBM hard disk. (Source: [Be07])

This means there are two ways to increase energy efficiency: always keep the HDD in the lowest power mode possible and avoid or reduce disk activity as much as possible.

The former approach is used in most modern Operating Systems. They use algorithms of varying complexity trying to predict the future disk accesses to find a good compromise between the best power mode for a given situation, the energy needed for the transition between the modes and the performance impact caused by added latency. The most widely used—and most simple—way is to use a user defined timeout after which the HDD is set to a lower power mode. This generic approach reduces the energy consumption in general, but by not taking the characteristics of the HDD and the access patterns into account it is far from optimal. Details of better strategies can be found in [Be07].

The latter approach—the reduction of disk activity—is the focus of this thesis. We will try to reduce energy consumption by actually reducing the amount of data written to the disk by compressing it on-the-fly.

2.2 Related Work

The idea to use data compression to reduce disk activity is not new, but it is usually utilized to improve performance without energy efficiency in mind.

2.2.1 Swap Compression

One of the first works to close the performance gap between main memory and disk drives was proposed by Douglas [Dou93]. His approach was to modify the paging algorithms of the virtual memory system: when a page has to be swapped out it is not directly written to the pagefile on the hard disk but instead compressed and stored in a *compression cache* in memory. Only when this cache is full are the oldest pages written back to disk. During swap in of a page the compression cache is first checked for that page. On a hit, the page would be taken from the cache and be decompressed for direct usage. Otherwise it would be read from disk as in a regular swap in strategy.

While this approach worked to reduce the amount of I/O operations it actually degraded performance of the tested applications. The relatively low available CPU processing power at the time was found to be the reason for that and he predicted that with increasing performance the results would be more favorable.

His idea of the compression cache was later taken up and improved on by Cortes and Becerra [CBC00]. Here the pages are also stored compressed on disk to increase the size of the swap area. Furthermore the disk accesses during swap in and out are grouped in order to no longer have the performance impact of disk seek times per page but only per group. Evaluation of the cache hit rates lead to the usage of two different paths for swap in and swap out which provides additional performance benefits by only keeping write buffers in the cache while read buffers bypass it.

2.2.2 Memory Compression

Compression can also be used to increase the amount of available memory by storing all data in it compressed and decompressing it on-the-fly when it is accessed. Early works like [MMS99] come to the conclusion that software compression is too slow and should only be used for swap compression like mentioned above. While they suggest that hardware compression is needed to be able to compress all data in memory without performance impact more recent approaches [ES05] are able to do this with a pure software implementation thanks to a specialized compression scheme coupled with a highly-efficient hierarchical memory layout.

Besides using the memory freed by compression to store more data, compression can also be used reduce the number of RAM modules needed to store the data. Obernolte [Obe03] improves energy efficiency by turning off the memory modules which no longer store data. More details about energy-aware memory compression can be found in [Wun04]. It compares the performance impact of several compression algorithms with their energy consumption and also evaluates which algorithms are best suited for the code and data segments of applications. The most efficient algorithm is found to be Lempel-Ziv-Oberhumer (lzo) [Obe02].

2.2.3 Compression for Energy Efficient Data Transmission

Another use of compression is to reduce the amount of data that has to be transferred when communicating. This is especially important for small sensor nodes where the most energy consuming task is the transmission of data over a wireless network. Work in this area has been done in [CF03] where compression is not only used to improve energy efficiency but also to reduce latency.

2.2.4 e2compr

e2compr [WWL] is an extension to the second extended filesystem (ext2) that allows transparent on-the-fly compression and decompression. It does not compress all the data in the file system but instead only the contents of regular files selected by the user. For data safety reasons, the administrative data like superblock, inodes and directory files are never compressed.

On an ext2 device without e2compr data is stored in fixed-size blocks (usually 1 kilobyte) and the inode holds (directly and possibly indirectly) a list of every block in a file. When reading data the file system code first determines the corresponding block it is stored in, gets the block, then reads the data from it. A special feature of ext2 is that files can contain holes which means that some blocks of a file are missing. In this case the apparent size of the file is much bigger than the product of the number of blocks it really holds and the block size. A missing block does not take up any space on the disk but is considered to be filled with zeroes.

Taking advantage of the missing blocks feature allows e2compr to add transparent compression and decompression without changing the way data is stored on disk: files are divided into clusters of n blocks (where n is constant for a given file) and e2compr replaces non-hole data in a cluster with blocks of compressed data followed by one or more missing blocks. The compression takes place whenever a cluster boundary is reached during writing. In some cases clusters do not compress well, for example when a cluster only contains one block of data or when the compressed data occupies more space than if uncompressed. In such cases the cluster is left unchanged. When a cluster is read it is necessary to differentiate between uncompressed and compressed clusters where the the holes actually contain data. This is done by indicating the missing blocks in a compressed cluster with FFFFFFFF and not with a zero pointer like the regular empty holes. Doing so reduces the maximum size of the file system by one block though.

The focus of e2compr is to increase disk space and to ensure data safety, not saving energy though. It nevertheless can be used as a starting point to see how much more energy the CPU needs to compress the data and if the HDD energy consumption can actually be reduced. The major drawbacks (both concerning energy savings and in general) of e2compr are:

- It is an extension of ext2 file system and thus incompatible with the journaling extensions of ext3 and naturally also any other file systems.
- Only file contents are compressed, but not the administrative data. In a file system with a big number of very small files this means that most data written to disk is uncompressed.
- The amount of data read from or written to disk can only be reduced for files bigger than one block since even a compressed cluster takes up at least one block.
- Data is only compressed when writing it to disk, there is no compressed buffer to reduce disk activity.

To solve those problems we designed the compressed block device *compr*.

Chapter 3

Design

3.1 Compressed Block Device

The main goal when designing the (virtual) compressed block device compr is to minimize the amount of HDD activity when reading and writing data whilst keeping the CPU load low by caching recently used uncompressed data. This is done to find a compromise between the additional energy consumed by the CPU and memory during compression and the energy saved with the HDD. compr works similar to a loop device by using a file or partition on a HDD as its backing store. All data written to compr is compressed and stored in a buffer in memory. Only when that buffer is full (or when the device is closed) is the data written back to disk. Increasing the available disk space or performance was not a design goal of compr.

As can be seen by the example of e2compr in Section 2.2.4 adding compression to an already existing file system can be difficult. Workarounds to differentiate compressed from uncompressed data need to be found and it is hard to achieve good compression ratio and high performance in a design that was never meant to support compression in the first place. A better solution would be to design a new file system from scratch to take full advantage of compression by focusing on its particular needs. Implementing a new file system is a daunting challenge though, and it does not solve the problem that it would be nice to have compression available independently from the used—and possibly specialized for a certain usage scenario—file system. Because of this we decided to implement compr as block device driver sitting between the file system and the disk as an additional layer to provide transparent compression for any file system. Since block device drivers are operating system dependent we chose to base our design on Linux and thus compr had to be designed with the architecture of the Linux block device layer in mind, but the design could be transferred to other operating systems as well.

3.1.1 Linux Block Device Layer

The block device layer in the Linux Kernel 2.6 is used to provide access to devices that transfer randomly accessible data in fixed-size blocks, that is disk drives primarily. Since disk drives normally are built with moving parts they have very high average access times. In a modern disk drive, a random access read or write requires about 10 milliseconds to complete, mainly because the read/write heads have to be moved into the right position above the disk surface and then have to wait for the part of the disk containing the data to arrive. However, once the heads are placed correctly and data is accessed sequentially a disk drive can sustain data rates of over 100 megabytes per second.

To minimize the performance impact of the high access times three techniques are used. Firstly, Linux implements a primary disk cache called *page cache* in main memory which is used when accessing data on a block device. This way on each cache hit the slow I/O operation on the disk drive can be avoided completely and the transfer rate is also improved since the page cache in memory is faster by an order of magnitude than any disk drive. Using direct I/O the page cache can be bypassed though which can be advantageous for applications like database systems which want to implement their own caching strategies in user mode. To further reduce the amount of head movement the smallest data unit that can be accessed on a block device is a 512-byte sector, ensuring that at least those 512 byte can always be read or written sequentially. Block devices can have bigger transferable units by combining sectors to blocks, the size of which is limited by the page size (usually 4 kilobytes). Any I/O operations smaller than that are automatically converted to an operation of the right size by the kernel. The last technique is the sorting of outstanding I/O requests in the command queue to try to have them all in the same direction of the head and the merging of multiple requests for contiguous sectors. Modern disk drives supporting Native Command Queueing (NQC) or Tagged Command Queueing (TQC) do not rely on the kernel for this anymore though as they can do it by themselves. The added latency induced by the reordering of the requests can reduce performance in rare situations though.

3.1.2 compr

In the simplest way a compressed block device could be designed by just adding it as an additional layer between the user and the disk drive like a loop device. Then on each write of a sector to the compressed block device the data would be compressed and written back to disk, and on a read the sector would be decompressed. As explained above the minimum size of an I/O operation on a block device is one sector though which means that no matter how well the sector can be compressed we always end up writing a whole 512-byte sector back to disk. This means alot of CPU overhead and thus energy consumption is added without actually reducing the work done by the disk drive at all.

To overcome this problem n sequential sectors have to be combined to blocks bigger than 512 byte and if that block compresses well only $m < n$ sectors have to be written to or read from disk. The bigger n is the higher is the possibility that a substantial number of sectors will not have to be accessed on disk at all. This is because most compression algorithms provide better results when data is split in only a few big partitions than many smaller ones. If n is too big CPU overhead becomes a serious problem though because now a big block has to be compressed/decompressed on each access to a sector. To mitigate this effect compr uses a two-tier cache hierarchy which can be seen in Figure 3.1.

3.1.3 Cache Hierarchy

In compr a variable (but fixed per device) number of n 512-byte sectors is combined into an uncompressed block (called *ublock* from now on). The default value of n is 32 resulting in ublocks of 16 kilobyte. On the first level of the cache hierarchy those ublocks are stored in a small least recently used (LRU) cache called *ucache*. This cache

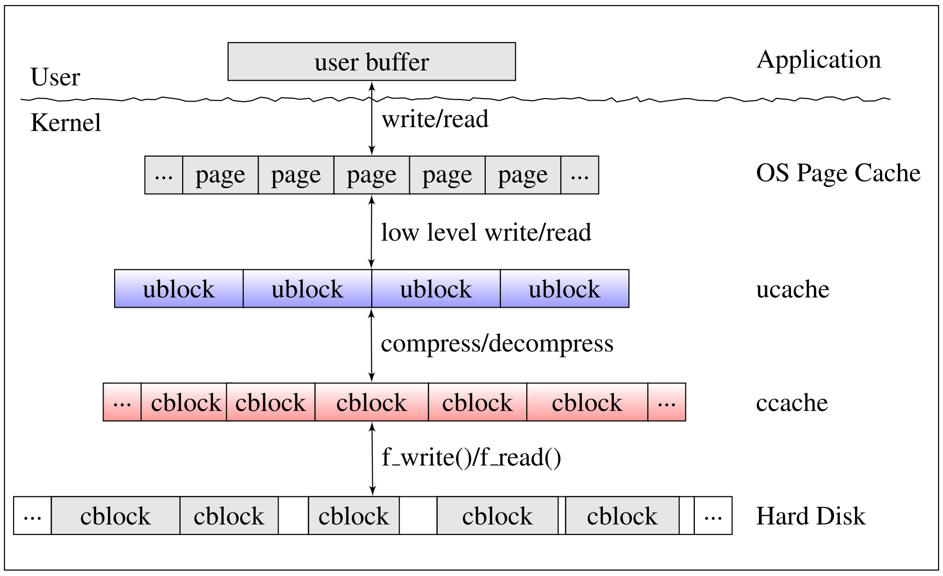
User
user buffer
Application
Kernel
write/read
... page page page page page ...
OS Page Cache
low level write/read
ublock ublock ublock ublock
ucache
compress/decompress
... cblock cblock cblock cblock cblock ...
ccache
f_write()/f_read()
... cblock cblock cblock cblock cblock ...
Hard Disk

Figure 3.1: The design of compr and its cache hierarchy.

helps to solve the problem of repeated access to sectors in the same ublock. The data only has to be decompressed on the first miss in the ucache, all subsequent requests to the same ublock can be satisfied with a simple memory copy. This specifically helps with sequential access as it reduces the number of decompressions needed from n to 1 compared to a design without a ucache.

Once the ucache overflows the oldest dirty entry is compressed into a compressed block (called *cblock*) and stored in compr's compressed disk cache. This big (the default size is 4 megabyte) second level *ccache* is the main technique used to reduce disk access as much as possible. Compared to a normal disk cache it differs in several points:

- As its entries are compressed it can potentially store much more data than normally would fit in main memory, resulting in a much more efficient use of available memory.
- The dirty cblocks are only written back to disk on a ccache overflow or when the device is closed. This means that disk access can be completely avoided if all the data fits in the ccache. The drawback is that this puts data safety at high risk if the ccache is lost because of unexpected shutdowns. This could be solved by writting back the dirty entries in intervalls like a normal disk cache does.
- The cblocks are not fixed-sized since they contain compressed data. This makes the implementation of the ccache more challenging because to make the best use of the available space, fragmentation has to be kept as low as possible. Similar problems are faced when designing dynamic memory allocators, so the approaches used there are applicable here too.

When there is no more room in the ccache the oldest dirty cblock is written back to disk. This way disk energy consumption is kept to a minimum compared to a system without compr as less data has to be stored on disk and the disk has to be accessed less often since there is more data stored in memory. The additional CPU energy consumption caused by compression and decompression is also kept relatively low by caching the ublocks too.

3.1.4 Uncompressed Cache ucache

The ucache used to store the ublocks is similar to a regular cache as it provides room for a fixed number of entries of a fixed size. Since we want most data to be compressed the ucache is fairly small (ranging from 2 to 8 ublocks) and thus simply iterating over all entries to find out if an access is a hit or a miss does not cause performance problems. As mentioned above, when the ucache is full an LRU replacement strategy is used and the oldest entry is written back if its dirty or discarded otherwise. A sequential read of all the blocks in a ublock will always result in a hit in the most recently used entry, so we decided to iterate over the ucache entries in reverse order of the LRU list.

The flow chart in figure 3.2 shows the steps taken in the ucache when data is written to compr.

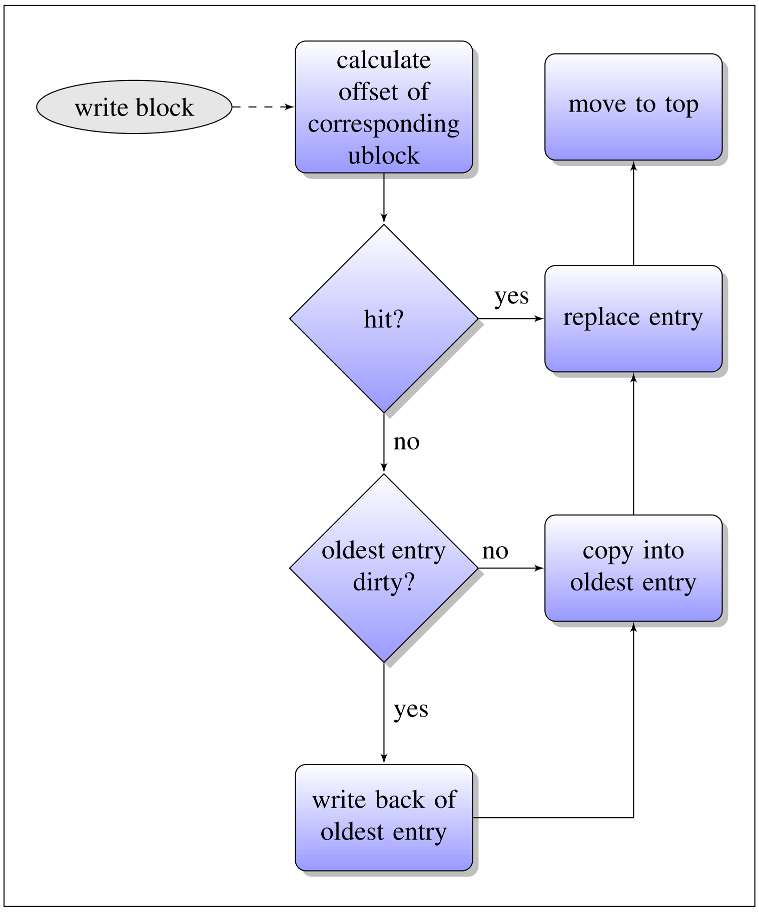

Figure 3.2: Flow chart of a write on compr.

3.1.5 Compressed Cache ccache

The ccache differs from the ucache by containing a variable number of entries of varying sizes. It also is considerably bigger and thus has room for a significantly bigger number of entries. This is aggravated by the fact that we are storing compressed data so even at the size of only 4 megabytes the ccache can potentially have tens of thousands of cblocks stored in it. If implented like the ucache the $O(n)$ Overhead for finding an entry could severely impact performance. We thus decided to base the lockup on a red-black-tree which offers an $O(log(n))$ search time.

Another problem of the big number of entries is the flushing of the cache when the device is closed. When writing all the dirty entries back to disk at least 512 byte need to be written for each cblock even if it is much smaller (see Section 3.1.1). If the page cache gets used for those writes we end up with a read and a write of 4 kilobytes per entry. This could easily nullify all the energy saved by the reduced number of disk accesses before the flush, especially taking the computational overhead of such a big number of small I/O operations into account. This can be mitigated by writing the blocks in ascending order to reduce the amount of disk seek time needed which is easy to do as the rb-tree sorts the entries anyway.

To improve the flush performance further we decided on another approach: instead of writing each dirty entry to disk the whole memory region containing the ccache is written to a reserved area on the disk. This way only one big I/O operation is needed. When the device is opened again the old ccache is read and the ccache is filled with its entries. This has the added advantage of providing a ccache that already contains the recently used blocks right when the device is opened. The drawbacks are that some disk space is lost to reserved area, and that restoring the ccache makes opening the device a computationally heavy operation which can be a big overhead if none of the blocks contained in the ccache are read. In general the much cheaper flush outweighs the slightly more complicated open though (see Table A.10).

Like in the ucache an LRU strategy is used to replace the oldest entries when the ccache is full. The steps taken in the ccache when a write back from the ucache occurs can be seen in Figure 3.3, the interaction of both caches can be seen in the flow chart in figure 3.5 which shows a read on compr.

3.1.6 Disk Layout

We decided to use a simple disk layout for compr which can be seen in figure 3.4. It starts with a short header containing all the information needed to open the device (size of the ublocks, the used compression algorithm and pointers of the old ccache needed to restore it). Following it is a reserved area which is used for the fast flush described above. This area cannot grow, so the size of the ccache is limited to the size used when the device first was created. The device can be recreated with a different ccache size but then all data stored on it is lost.

The next part is the *block allocation table* (BAT). It contains the size of each block saved on the disk. A direct mapping is used (the first entry is the size of the first cblock and so on) for fast access and small memory footprint. When the device is opened the BAT is loaded into memory and used for all following accesses. It is updated whenever the size of a cblock changes (due to a write) and when the device is closed the BAT is written back to disk. A size of 0 bytes indicates that a cblock does not contain any data. We take advantage of that by not accessing the disk at all when a cblock of size 0 is accessed and instead just return a buffer filled with zeroes. If the size of a cblock

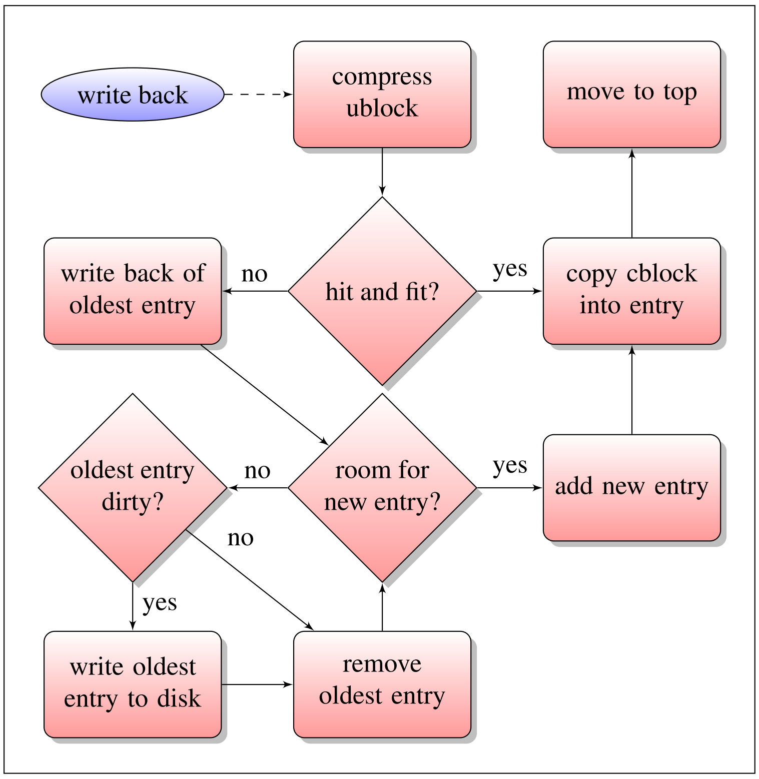

Figure 3.3: Flow chart of a write back from ucache to ccache.

is as big as a ublock it indicates that the cblock is not compressed. This can happen when the data does not compress well and actually takes up the same or more space in compressed form than uncompressed.

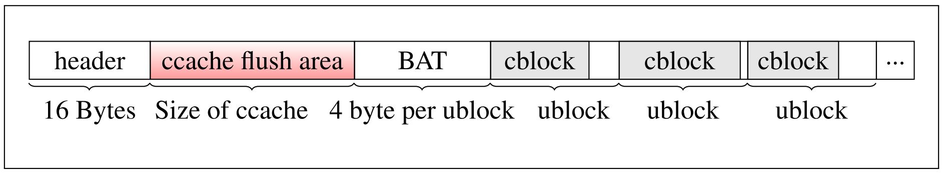

Figure 3.4: The disk layout used in compr.

After the BAT the data area begins. Here all the cblocks are stored in ascending order, each one starting at the offset of the corresponding ublock. Since a cblock can never be bigger than a ublock this ensures that there is always enough room for the cblocks. Disadvantages of this are a high degree of internal fragmentation caused by all the gaps between cblocks. Those gaps could be used to increase the available disk space by compacting the cblocks but that is beyond the scope of this thesis.

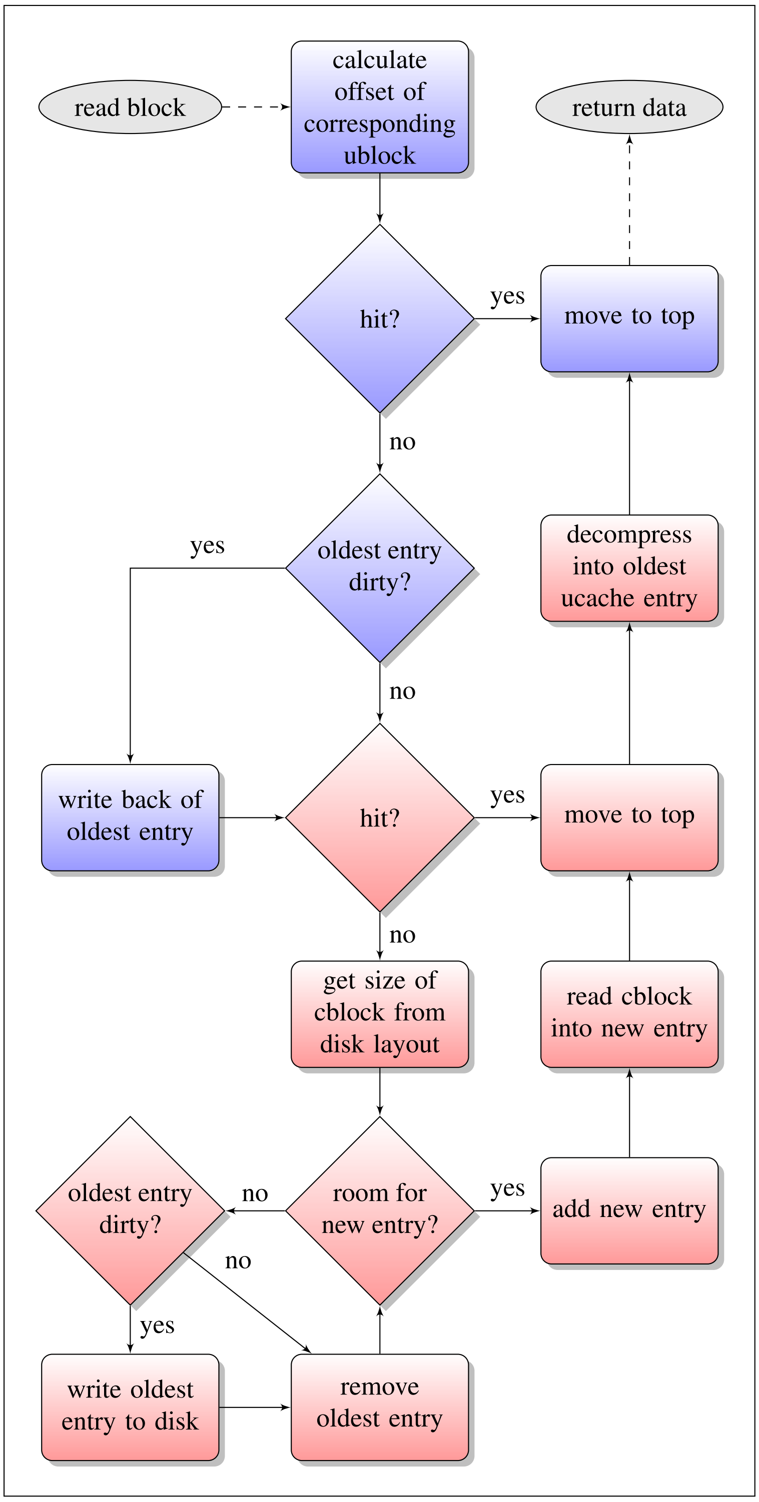
read block
calculate offset of corresponding ublock
return data
hit?
yes
move to top
no
oldest entry dirty?
yes
decompress into oldest ucache entry
no
write back of oldest entry
hit?
yes
move to top
no
get size of cblock from disk layout
read cblock into new entry
oldest entry dirty?
no
room for new entry?
yes
add new entry
no
yes
write oldest entry to disk
remove oldest entry

Figure 3.5: Flow chart of a read on compr. Red nodes are part of the ccache, blue nodes of the ucache.

Chapter 4

Implementation

To evaluate if compr is able to improve energy efficiency we implemented it as a block device driver for the Linux operating system. The kernel version used was 2.6.11 [Tor08]. This system was chosen because it is open source and well documented which facilitates the implementation and also because it is available for embedded systems too.

4.1 Compression Algorithms

The compression and decompression of data is at the core of compr. Energy consumption can only be reduced if the workload caused by the compression plus the additional overhead of the cache hierarchy is lower than the energy saved by the reduced disk activity. Because of this, compr has an open interface which allows to easily add more or to select the best compression algorithm for a given system to meet this requirement. The following five algorithms were examined in this thesis.

4.1.1 gzip

Gzip is based on the DEFLATE [Deu96] algorithm which combines LZ77 and Huffman coding. LZ77 is a dictionary based algorithm which scans the data for redundancy using a sliding window and performs a run-length encoding. It offers an adjustable compression-level vs. speed of encoding ratio, with gzip1 being the lowest compression and gzip9 the best. This makes it suitable for finding the right balance to save energy.

Another advantage of it is the wide availability since zlib, which implements the DEFLATE algorithm, is part of the Linux kernel.

4.1.2 lzrw3a and lzv1

Like gzip, the lzrw3a [Wil91] algorithm is also based on LZ77. Compared to gzip it offers slightly worse compression but is significantly faster. It also has a very small memory footprint, requiring only 16 kilobyte of memory for both compression and decompression.

LZV1 is similar to LZRW3A but is tuned for even faster performance. In most cases it is not that much slower than a pure memory copy, but it also only provides very low compression ratios.

4.1.3 lzo

One of the fastest compression algorithms based on LZ77 is lzo [Obe02]. Its compression ratio is equal to or better than that of gzip while being moderately faster during compression and significantly faster at decompression using less than a tenth of CPU time in comparision.

4.1.4 bzip2

Contrary to the other algorithms mentioned so far, bzip2 is not based on LZ but on the Burrows-Wheeler transform [BW94]. Its compression stack is quite complex, including amongst others two run-length encodings, the Burrows-Wheeler transform, sorting of the input data and a Huffman coding. This makes compression and decompression quite slow, but the compression ratio obtained by bzip2 surpasses that of gzip by up to 65%, especially when compressing text files. As it is an asymetric algorithm decompression is relatively fast compared to compression though, so it might offer good performance for data that is rarely written to but often read.

4.2 Implementation of compr

The implementation of compr is similar to that of the loop device driver included in the Linux kernel. It consists of two parts: a small user space tool called *comprsetup* which is used to create the device and the block device driver itself in kernel space.

When creating the device comprsetup allows the user to specify which compression should be used, how many blocks a cblock contains, how big the caches are and which device is to be used as backing file. This information along with a file handle of the backing file is then passed to the driver. This has to be done because if the backing file was opened in the driver it would belong to whichever process accessed the driver during the open operation. As this could be any process accessing the device compr would not be transparent for that process anymore because its file descriptor table would be modified. This could cause problems because the number of file descriptors per process is limited.

In the driver the file handle is then used to read the header of the backing file and the BAT is copied into memory. The Linux file operations are meant to be accessed from user space though and expect user space buffers whereas all the memory allocated by the driver is in kernel space. To circumvent this problem the current process address limits have to be modified for every read and write which is easy to do using the `set_fs()` function.

A more serious problem is that all the disk accesses by the driver are also buffered in the page cache and thus we get an additional layer in the cache hierarchy described in Section 3.1.3. This second page cache could be bypassed by using direct I/O, but direct I/O is optimized for usermode applications and does not work from within the kernel. Making it work with kernel buffers would need substantial changes to the direct I/O implementation in the kernel which was beyond the scope of this thesis.

To keep the time spent in the read and write operations of the device low and to not block applications for too long all compression and decompression is done in a seperate kernel thread. Using more than one thread for this on systems with more than one CPU could increase the performance at the cost of additional synchronization overhead. As our fucus is on energy consumption and not performance though we decided to use the single threaded approach.

4.2.1 Implementation Details of the ucache

Since the ucache has a fixed size it is implemented as a single memory area (allocated with `vmalloc()`) containing all its entries. They are linked together by a singly-linked list and to implement the LRU replacement strategy the last entry is moved to the top of the list on each access.

4.2.2 ccache Implementation Details

The implementation of the ccache is based on the red-black-tree implementation included in the kernel for fast lockup and an LRU list like the one used in the ucache. Because the ccache entries do not have a fixed size we cannot allocate all its memory as one block and then divide it equally, instead we have to allocate memory per entry. The memory allocation functions offered by the kernel are not suited for a big number of allocations of varying sizes though: `kmalloc()` is optimized for small sized memory allocations and suffers from fragmentation on non power-of-two sizes, `vmalloc()` is meant for bigger allocations and has significant fragmentation when used for small sizes. To be able to write back the ccache with just one I/O operation when flushing we furthermore need all its entries to be in one memory area which is not possible with either of those two functions.

Our solution to this problem is to use *Two Level Segregate Fit* (TLSF) [MRC03], a general purpose dynamic memory allocator designed to meet real-time requirements. It is much faster than the memory allocations in the kernel while being very memory efficient thanks to the low amount of fragmentation. It also can be used to allocate all its memory within a big memory area previously allocated by `vmalloc()` and thus fulfills all our requirements.

To be able to write all of the ccache to disk and then restore it later on not only its data needs to be written though but also a link to all the entries to be able to find them in the memoy block. Instead of writing a seperate index to disk we decided to use the following approach: We not only use the memory area managed by TLSF to store the cblocks but also put the management information (pointers to the linked list and the red-black-tree, dirty bit and the offset of the cblock) of the entries in it. The data and the information are allocated together with one operation to keep the overhead low, and since the data directly follows the information we do not need a pointer to the data so we can save some memory.

When restoring this memory block from disk the only additional information we need are the offset of the first entry from the start of the memory block and the address of the memory block which are saved in the header. We can then access the first entry and all subsequent entries by simply adjusting the links in them with the saved address and put all the cblocks into our new ccache. Since we are not allocating the cblocks in the same order they were allocated before the flush it is possible to have more fragmentation though, so all dirty cblocks which do not fit in the new ccache are written back to disk.

Chapter 5

Evaluation and Discussion

To evaluate if and by how much file compression can improve energy efficiency and which compression algorithm is best suited for this task we measured the energy consumption in several scenarios. In this chapter only parts of the measurements will be presented, tables with all obtained results can be found in Appendix A.

In order to avoid any kind of external influence on the obtained results all unneeded and potentially energy consuming background tasks were closed. Furthermore, the test systems were rebooted before each test run to set them into the same initial state each time and to empty the page cache in particular. The already existing power management features of the Linux kernel—especially the CPU throttling mechanisms —were not disabled since they would not be disabled in a real production system striving for low energy consumption either.

5.1 Test Systems

Two systems were used for the evaluation: a desktop PC and an embedded system. The desktop system has a 1.5 GHz Intel Pentium 4 [Int03] processor and 768 megabyte RAM. We used a 40 gigabyte Samsung Spinpoint MP0402H [Sam] 2.5 inch notebook HDD connected to the mainboard's IDE controller. Both CPU and HDD energy consumption were measured to see if the energy saved by the disk drive outweighs the CPU overhead caused by the compression.

The embedded system is a Triton Starter Kit III [KAR] evaluation board with a 312 MHz Intel XScale PXA270 [Int] processor and 64 megabyte of main memory. A 4 gigabyte Hitachi microdrive [Hit] connected to the Compact Flash interface was used as the disk drive. As it is a System-on-a-chip design it was not possible to measure HDD and CPU energy consumption seperately, so we focused on total system energy consumption instead.

5.2 Compressed File System

Our first evaluations were done using e2compr to see if a compressed file system not designed for energy efficiency can be used to save energy, and to get a general idea of the HDD to CPU energy consumption ratio. The latest e2compr version (0.4.51) was used on the desktop system, but as the embedded system was running an older Linux kernel we had to use e2compr version 0.4.45 on it.

5.2.1 Desktop PC

To see if the compression helps with energy consumption of the desktop system at all our first tests were a best case scenario: reading and writing a 250 megabyte file containing only zeroes to an e2compr filesystem. This file compresses extremely well and thus results in the lowest disk activity.

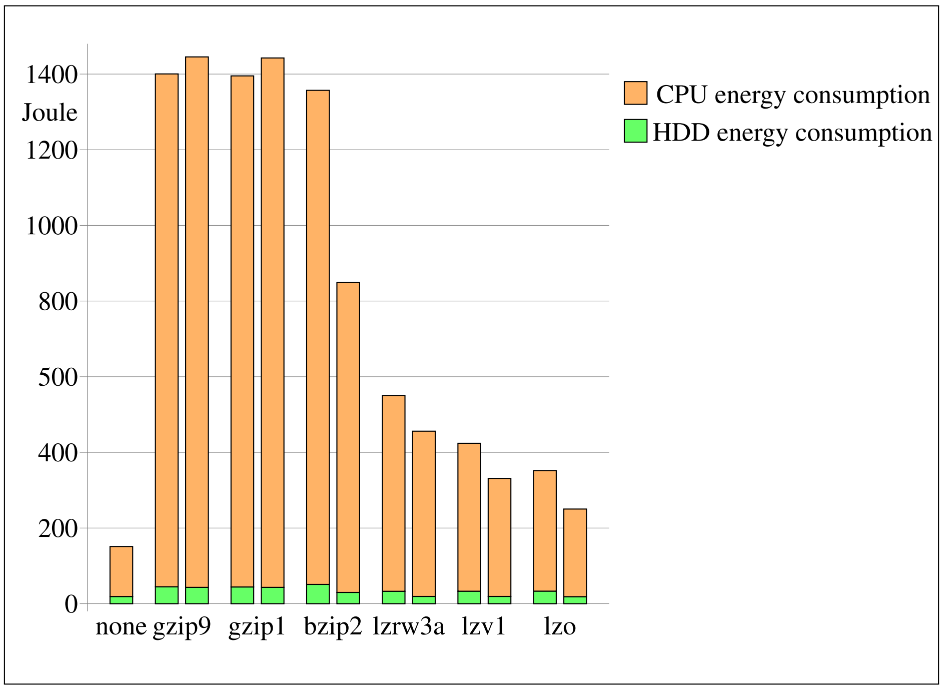

Figure 5.1: Energy consumption when writing 250 MB of zeroes on the desktop system using e2compr. The left bar is using a cluster size of 4 kilobyte, the right bar 8 kilobyte.

The energy consumption when writing can be seen in Figure 5.1. Even with the best performing compression algorithm (LZO) 60% more energy is needed compared to writing the file to an uncompressed file system. As the chart shows we are actually able to improve HDD energy efficiency by a small amount, but the total energy consumption is clearly dominated by the CPU.

There are two main reasons for the increased energy consumption: firstly, during the uncompressed write the CPU is idle all the time, consuming only about 10 W. When compressing it operates at full load though and the energy consumption increases to more than 60 W. As the HDD energy consumption only decreases by 2 W when there is no disk activity energy can only be saved if the compressed write is significantly shorter. Secondly, the write rate of the HDD is higher than the compressed data throughput of the CPU. This means no matter how well the data compresses the write operation will always take longer than the uncompressed write, making it impossible to save energy in this scenario.

The results when reading are similar, showing a 45% increase in the best case. To see if we could save energy on a busy system where the CPU is at full load even without compression we decided to compile the Linux kernel on an e2compr file system. The results (see Table A.3) improve but the energy consumption still increases by 8%.

5.2.2 Embedded System

The best case scenario was tested on the embedded system too, but because of the slower disk a smaller 25 megabyte file of zeroes was used. Figure 5.2 shows the obtained results for both reading and writing.

Using lzrw3a, lzv1 or lzo and a cluster size of 8 kilobyte the system energy consumption actually can be reduced to as low as 65%. That is because the CPU of the embedded system consumes a similar amount of energy as the HDD and thus we do not face the same problems as with the desktop system. Additionally the CPU can decompress data faster than the disk can read uncompressed data causing the compressed read to take less time.

When writing the energy consumption changes drastically, resulting in a 339% increase in the best case. A small part of this is caused by the computationally more expensive compression compared to decompression, but the main reason is that we used an older version of e2compr. Instead of compressing and writing a cluster directly to disk, the file first is completely written as an uncompressed file, then each cluster is read in again, gets compressed and then written back to disk. Thus thrice the work has to be done which explains the tripled energy consumption.

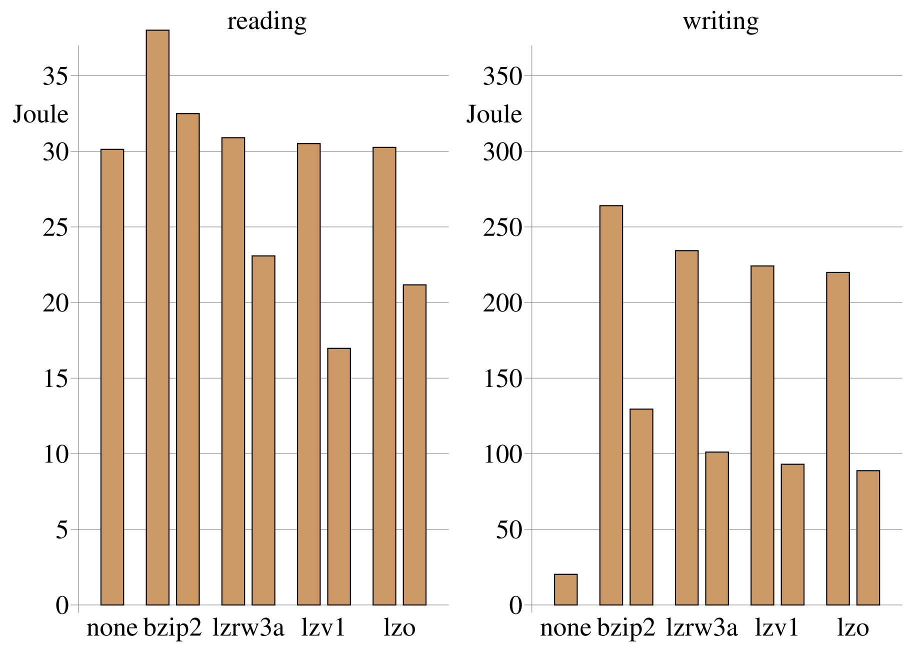

Figure 5.2: System energy consumption on the embedded system using a 25 megabyte file of zeroes on e2compr. The left bar is using a cluster size of 4 kilobyte, the right bar 8 kilobyte.

5.3 Compressed Block Device

As shown above compression cannot be used to improve the energy efficiency of the desktop system which is why we tested our implementation of compr on the embedded system only. All measurements were run with 16 kilobyte ublocks, a 4 megabyte ccache and the ucache was set to contain a maximum of 4 entries. These values were selected as they provide good results and also leave enough memory for other applications.

5.3.1 Embedded System

Our first test for compr was the best case scenario we also used for e2compr: reading and writing a 250 megabyte file of zeroes. The results can be seen in Figure 5.3 where 'ext2' stands for a regular uncompressed ext2 file system and 'none' is compr with no compression, that is the cblocks are not compressed and the same size as the ublocks.

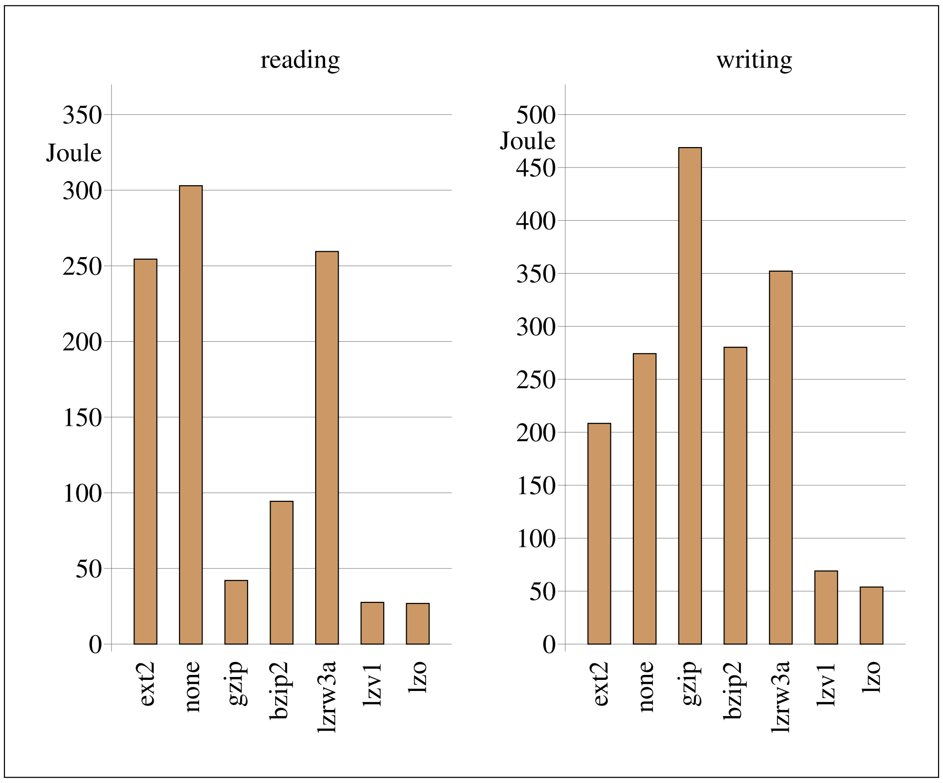

Figure 5.3: System energy consumption on the embedded system when reading and writing a 250 megabyte file of zeroes using compr.

Thanks to the compressed cache energy efficiency is much increased when reading for most compression algorithms. That is because the small 4 megabyte ccache actually contains all data of the 250 megabyte file and thus no data at all has to be read from disk. The decompression from the ccache to memory is also much faster than reading from disk which further reduces energy consumption by making the read operation shorter. So not only do we improve energy efficiency but also performance. The best performing algorithm here again is lzo by reducing energy consumption to 11%. lzrw3a fails to deliver good results because it does not compress the ublocks as well as the other algorithms and thus its ccache contains less than 33 megabyte of uncompressed data.

When writing 250 megabyte of zeroes only lzv1 and lzo can reduce the energy consumption. The gzip and bzip2 algorithms are too computationally expensive when compressing data and lzrw3a suffers from the bad compression ratio mentioned above.

Also observable is the higher energy consumption when reading (119%) and writing (134%) to compr without using compression. This overhead is caused by the additional memory copies between the caches. When writing data it is first copied to the page cache, from there to the ucache, then the ccache, the page cache again and finally written to disk. As mentioned in 4.2 this overhead could be reduced if it was possible to bypass the second page cache using direct I/O. It is remarkable that compr can improve energy efficiency when compressing data despite this relatively big overhead.

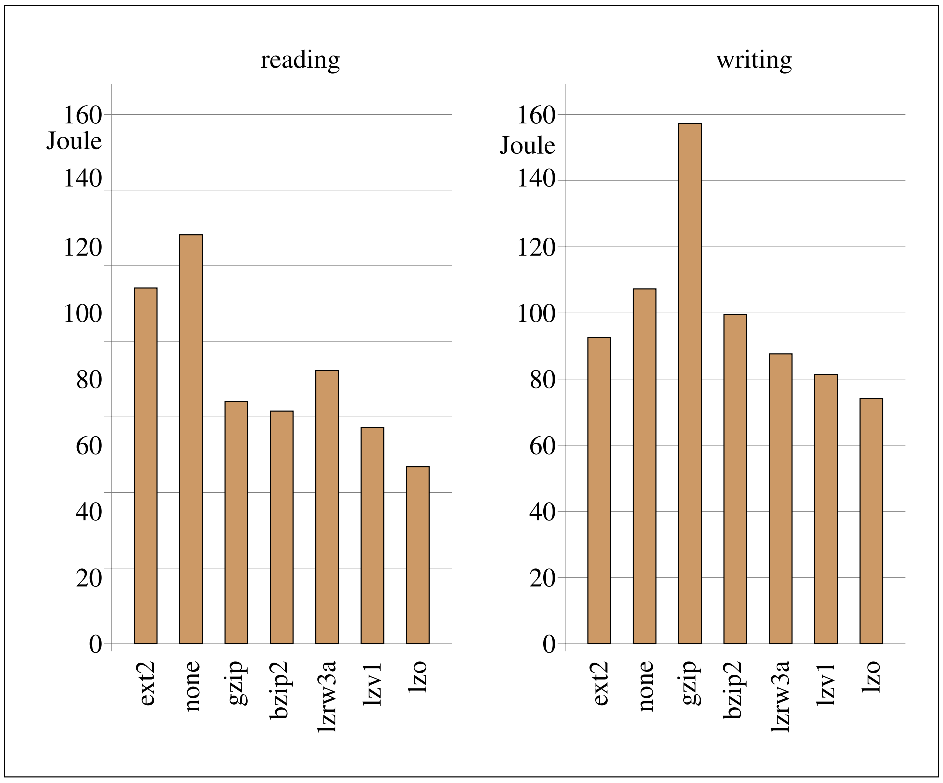

Figure 5.4: System energy consumption on the embedded system when reading and writing 45 megabyte of Linux source code using compr.

To simulate a more realistic scenario than our best case we used a file containing the first 45 megabyte of the (uncompressed) Linux source code. For the write measurements this file was first copied to a RAM disk and then written to disk from there. This was necessary because the disk we used was the only storage device available on the embedded system and transfering the file over the network turned out to be a bottleneck.

As can be seen in Figure 5.4 compr can improve efficiency in this scenario too by reducing energy consumption to 50% when reading and 80% when writing. Again lzo provides the best result because it provides the best trade-off between compression ratio and compression speed. The data is compressed to about 10% smaller cblocks by bzip2 but the compression takes disproportionately more energy and time resulting in a higher total energy consumption. lzrw3a and lzv1 provide decent results too but lzo is superior to them in any way and thus the compression algorithm of choice. gzip is both too slow and does not compress well enough to achieve good results.

Chapter 6

Summary

With the increasing amount of data that needs to be stored, hard disk drives are becoming one of the major energy consumers in many computer systems. This is intensified by the improved energy efficiency of many other system components whereas little to no work has been done to reduce the energy consumption of disk drives.

Our proposed solution is to transparently compress all data that is to be written to disk. This allows us to store more data in main memory and thus the number of I/O operations is reduced, and since less data has to be read and written the length of the disk accesses are shortened. Due to this reduction of disk activity the energy efficiency of the hard disk is improved.

6.1 Conclusion

As shown by our evaluations the reduced disk energy consumption does not always translate to an improved energy efficiency for the whole system though. The reason for this is that the additional energy needed by the CPU for compression can greatly exceed the energy saved by the disk drive.

The CPUs used in regular desktop systems today consume more energy when idling than a disk drive under full load, and the disk data transfer rates are higher than the throughput of compressed data of the CPU. This results in slower and longer I/O operations when using compression and thus no energy can be saved. This might change in the future if the trend to faster and more energy efficient CPUs continues.

On embedded systems where the CPU and HDD energy consumption are similar our approach is able to reduce the system energy consumption to 50% when reading and 80% when writing. Here the compressed cache in memory also has a bigger impact since embedded systems usually have a very limited amount of RAM. Furthermore we are also able to improve performance because the small embedded hard drives only have low data transfer rates and the compression enables us to write more data in less time.

In general the potential to save energy with our approach increases with faster but highly energy efficient CPUs and higher HDD energy consumption, which seems to be the direction for both CPU and HDD development.

6.2 Future Work

One way to improve the efficiency of our compression layer would be the reduction of overhead by bypassing the second page cache using direct I/O. This would need substantial changes to the direct I/O implementation in the Linux kernel to be able to use it not only from user mode but also kernel mode. Even more overhead could be eliminated if the first page cache could be bypassed too, but this would need changes to the user mode applications.

Another improvement would be to use a better disk layout similar to the one used by the ccache to reduce the amount of unused disk space. Compacting the compressed data and thereby increasing the available space might result in less needed HDDs and thus quite a big energy conservation.

With flash memory based Solid State Disks (SDDs) becoming more and more popular and already replacing HDDs in some low energy devices it should be checked if their energy efficiency can also be improved by compression. In general they consume less energy than disk drives because they do not contain any moving mechanical parts. They still need more energy for reading and writing than for idling though, so a reduction of disk activity should still be able to reduce energy consumption too, just to a lesser extent than with disk drives.

Appendix A

First Appendix

Following are all the details of the obtained results of the evaluations described in Section 5. The 'Relative' column describes the energy consumption relative to the uncompressed case, the lower the value the higher the energy efficiency.

Algorithm	Cluster Size	HDD	CPU	Total	Time	Relative
none	-	19.01 J	132.38 J	151.38 J	7621 ms	100%
gzip9	4 kB	44.77 J	1355.27 J	1400.04 J	25847 ms	925%
	8 kB	42.54 J	1403.97 J	1446.51 J	26182 ms	956%
gzip1	4 kB	44.23 J	1350.90 J	1395.13 J	25456 ms	921%
	8 kB	42.33 J	1400.36 J	1442.69 J	26173 ms	953%
bzip2	4 kB	51.03 J	1305.99 J	1357.02 J	30002 ms	896%
	8 kB	29.88 J	818.99 J	848.86 J	18000 ms	561%
lzrw3a	4 kB	32.96 J	514.46 J	550.41 J	17992 ms	364%
	8 kB	19.26 J	436.87 J	456.14 J	11093 ms	301%
lzv1	4 kB	32.94 J	391.00 J	423.94 J	18070 ms	280%
	8 kB	19.11 J	312.06 J	331.18 J	11020 ms	218%
lzo	4 kB	33.25 J	318.68 J	351.98 J	17582 ms	233%
	8 kB	18.53 J	231.71 J	250.23 J	10577 ms	165%

Table A.1: Energy consumption when writing 250 megabyte of zeroes on the desktop system using e2compr.

Algorithm	Cluster Size	HDD	CPU	Total	Time	Relative
none	-	20.50 J	164.85 J	185.35 J	8534 ms	100%
gzip9	4 kB	20.49 J	250.84 J	271.34 J	8757 ms	146%
	8 kB	26.33 J	262.48 J	288.82 J	11348 ms	156%
gzip1	4 kB	25.87 J	262.88 J	288.75 J	11143 ms	156%
	8 kB	20.62 J	253.35 J	273.96 J	8838 ms	148%
bzip2	4 kB	35.41 J	637.34 J	672.75 J	15376 ms	339%
	8 kB	56.88 J	522.50 J	579.38 J	24734 ms	313%
lzrw3a	4 kB	20.61 J	252.31 J	272.92 J	8828 ms	147%
	8 kB	25.14 J	258.22 J	283.36 J	10824 ms	150%
lzv1	4 kB	20.53 J	257.29 J	277.82 J	8786 ms	150%
	8 kB	29.23 J	268.35 J	315.58 J	12539 ms	170%
lzo	4 kB	20.54 J	236.18 J	256.72 J	8795 ms	139%
	8 kB	24.93 J	240.84 J	265.77 J	10739 ms	143%

Table A.2: Energy consumption when reading 250 megabyte of zeroes on the desktop system using e2compr.

Algorithm	Cluster Size	HDD	CPU	Total	Realtive
none	-	324.35 J	11737.2 J	12061.6 J	100%
gzip9	4 kB	442.94 J	13188.6 J	13631.5 J	113%
	8 kB	500.47 J	15025 J	15525,5 J	128%
gzip1	4 kB	430.66 J	13106.6 J	13537.3 J	112%
	8 kB	458.35 J	14987.7 J	15473.1 J	128%
bzip2	4 kB	570.37 J	17259.3 J	17829.7 J	147%
	8 kB	620.35 J	16573.4 J	17193.7 J	143%
lzrw3a	4 kB	435.32 J	13357.1 J	13792.4 J	114%
	8 kB	423.49 J	13457.4 J	13881.0 J	115%
lzv1	4 kB	419.62 J	13434.4 J	13854.0 J	114%
	8 kB	406.35 J	13067.2 J	13473.5 J	112%
lzo	4 kB	410.73 J	12938.7 J	13349.4 J	111%
	8 kB	390.36 J	12634.4 J	13024.8 J	108%

Table A.3: Energy consumption when compiling the Linux kernel on an e2compr file system on the desktop system.

Algorithm	Cluster Size	Energy	Time	Relative
none	-	30.13 J	12498 ms	100%
bzip2	4 kB	38.01 J	15297 ms	126%
	8 kB	32.49 J	8604 ms	108%
lzrw3a	4 kB	30.90 J	12762 ms	103%
	8 kB	23.08 J	9570 ms	77%
lzv1	4 kB	30.51 J	12727 ms	101%
	8 kB	19.47 J	8030 ms	65%
lzo	4 kB	30.26 J	12588 ms	100%
	8 kB	21.17 J	8773 ms	70%

Table A.4: Energy consumption when reading 25 megabyte of zeroes on the embedded system using e2compr.

Algorithm	Cluster Size	Energy	Time	Relative
none	-	20.23 J	8277 ms	100%
bzip2	4 kB	264.07 J	104606 ms	1305%
	8 kB	129.52 J	51424 ms	640%
lzrw3a	4 kB	234.28 J	93514 ms	1158%
	8 kB	101.10 J	40643 ms	500%
lzv1	4 kB	224.20 J	89512 ms	1108%
	8 kB	93.07 J	37486 ms	460%
lzo	4 kB	219.88 J	87634 ms	1087%
	8 kB	88.79 J	35735 ms	439%

Table A.5: Energy consumption when writing 25 megabyte of zeroes on the embedded system using e2compr.

Algorithm	ccache entries	Energy	Time	Relative
ext2	-	208.40 J	81457 ms	100%
none	255	274.25 J	108292 ms	134%
gzip	17054	468.78 J	246385 ms	225%
bzip2	17054	280.26 J	147547 ms	134%
lzrw3a	2103	352.16 J	143421 ms	168%
lzv1	16420	69.05 J	36813 ms	33%
lzo	17054	53.23 J	26272 ms	25%

Table A.6: Energy consumption when writing 250 megabyte of zeroes on the embedded system using compr.

Algorithm	ccache entries	Energy	Time	Relative
ext2	-	254.42 J	104388 ms	100%
none	248	302.95 J	122774 ms	119%
gzip	17054	42.05 J	22157 ms	17%
bzip2	17054	64.38 J	45883 ms	37%
lzrw3a	2094	259.48 J	105089 ms	102%
lzv1	16354	27.59 J	14130 ms	11%
lzo	17054	26.87 J	14121 ms	10%

Table A.7: Energy consumption when reading 250 megabyte of zeroes on the embedded system using compr.

Algorithm	ccache entries	Energy	Time	Relative
ext2	-	92.59 J	39668 ms	100%
none	248	107.23 J	41385 ms	115%
gzip	517	157.24 J	45647 ms	169%
bzip2	751	99.54 J	40368 ms	108%
lzrw3a	576	87.63 J	37617 ms	95%
lzv1	631	81.46 J	33679 ms	88%
lzo	682	74.11 J	31254 ms	80%

Table A.8: Energy consumption when writing 45 megabyte of the Linux source code on the embedded system using compr.

Algorithm	ccache entries	Energy	Time	Relative
ext2	-	107.55 J	42548 ms	100%
none	248	123.62 J	45328 ms	115%
gzip	521	73.19 J	32147 ms	68%
bzip2	752	70.33 J	34261 ms	65%
lzrw3a	584	82.62 J	36217 ms	77%
lzv1	637	65.35 J	28384 ms	61%
lzo	685	53.50 J	21723 ms	50%

Table A.9: Energy consumption when reading 45 megabyte of the Linux source code on the embedded system using compr.

Flush	ccache entries	Energy Flush	Energy Open	Total	Relative
normal	17054	340.52 J	2.86 J	343.38 J	100%
ordered	17054	310.65 J	2.73 J	313.35 J	91%
fast	17054	1.87 J	7.84 J	9.71 J	3%

Table A.10: Comparision of the different ccache flushes.

Bibliography

[Be07] Frank Bellosa and Andreas Weißel. Power management lecture, 2007. http://i30www.ira.uka.de/teaching/courses/lecture.php?courseid=151&lid=en.

[BW94] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, 1994.

[CBC00] Toni Cortes, Yolanda Becerra, and Raúl Cervera. Swap compression: resurrecting old ideas. *Softw., Pract. Exper.*, 30(5):567–587, 2000.

[CF03] Mo Chen and Mark L. Fowler. The Importance of Data Compression for Energy Efficiency in Sensor Networks. In *Conference on Information Sciences and Systems, The Johns Hopkins University*, March 2003.

[Deu96] L. Peter Deutsch. RFC 1951: DEFLATE compressed data format specification version 1.3, May 1996. http://tools.ietf.org/html/rfc1951.

[Dou93] Fred Douglis. The compression cache: Using on-line compression to extend physical memory. In *Proceedings of the USENIX Winter Technical Conference*, pages 519–529, January 1993.

[ES05] Magnus Ekman and Per Stenstrom. A robust main-memory compression scheme. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 74–85, Washington, DC, USA, 2005. IEEE Computer Society.

[Hit] Hitachi. Microdrive 3K4. http://www.hitachigst.com/tech/techlib.nsf/products/Microdrive_3K4.

[Int] Intel. XScale PXA 270 Processor. http://www.intel.com/design/embeddedpca/applicationsprocessors/302302.htm.

[Int03] Intel. Intel Pentium 4 Processor, 2003. http://www.intel.com/products/processor/pentium4/index.htm.

[KAR] KARO. Starter-Kit III für TRITON 270. http://www.karo-electronics.de/stk3.html.

[MMS99] Kjelso M., Gooch M., and Jones S. Performance Evaluation of Computer Architectures with Main Memory Data Compression. In *Journal of Systems Architecture. Vol. 45*, pages 571–590, 1999.

[MRC03] M. Masmano, I. Ripoll, and A. Crespo. Dynamic storage allocation for real-time embedded systems, 2003.

[Obe02] Markus F.X.J. Oberhumer. Lzo version 1.08, Jul 2002. http://www.oberhumer.com/opensource/lzo/.

[Obe03] Jürgen Obernolte. Energy-Aware Memory Management, April 2003. http://www4.informatik.uni-erlangen.de/DA/pdf/DA-I4-2003-04-Obernolte.pdf.

[Sam] Samsung. Spinpoint M series MP0402H. http://www.samsung.com/me/products/hdd/25inchmobile/mp0402h.asp.

[Tor08] Linus Torvalds. The Linux Kernel Archives, 2008. http://www.kernel.org.

[Wik08] Wikipedia. Hard disk drive, 2008. http://en.wikipedia.org/wiki/Hard_disk_drive.

[Wil91] Ross Williams. Notes on the LZRW3-A algorithm, Jul 1991. http://www.ross.net/compression/download/original/old_lzrw3-a.txt.

[Wun04] Holger Wunderlich. Energy aware memory compression, October 2004. http://www4.informatik.uni-erlangen.de/SA/pdf/SA-I4-2004-xx-Wunderlich.pdf.

[WWL] Matthias Winkler, Paul Whittaker, and Terry Loveall. e2compr - transparent compression for the ext2 filesystem. http://e2compr.sourceforge.net/.