

**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**  
BRNO UNIVERSITY OF TECHNOLOGY

**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**  
**ÚSTAV INFORMAČNÍCH SYSTÉMŮ**

FACULTY OF INFORMATION TECHNOLOGY  
DEPARTMENT OF INFORMATION SYSTEMS

**COMPRESSION FILTER FOR REDIRFS FRAMEWORK**

**BAKALÁŘSKÁ PRÁCE**  
BACHELOR'S THESIS

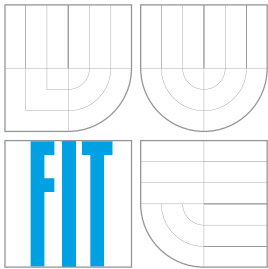
**AUTOR PRÁCE**  
AUTHOR

**JAN PODROUŽEK**

BRNO 2007



**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**  
BRNO UNIVERSITY OF TECHNOLOGY



**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**  
**ÚSTAV INFORMAČNÍCH SYSTÉMŮ**

FACULTY OF INFORMATION TECHNOLOGY  
DEPARTMENT OF INFORMATION SYSTEMS

# **KOMPRESNÍ FILTR PRO REDIRFS**

COMPRESSION FILTER FOR REDIRFS FRAMEWORK

**BAKALÁŘSKÁ PRÁCE**  
BACHELOR'S THESIS

**AUTOR PRÁCE**  
AUTHOR

**JAN PODROUŽEK**

**VEDOUCÍ PRÁCE**  
SUPERVISOR

**Ing. TOMÁŠ KAŠPÁREK**

BRNO 2007

## Abstrakt

Tato práce se v první části zabývá problematikou vývoje modulu pro linuxové jádro, specificky moduly implementující filtry pro RedirFS. V následující části zvažuje různé druhy kompresních algoritmů a implementační detaily CryptoAPI rozhraní linuxového jádra. Dále popisuje návrh a fungování CompFLT modulu pro linuxové jádro implementující kompresní filtr nad systémem RedirFS. Srovnávací testy provedené při použití CompFLT modulu jsou prezentovány v závěrečné části.

## Klíčová slova

linux, jádro, modul, komprese, redirfs, souborový systém

## Abstract

This thesis in its first part examines the problems of developing linux kernel modules, specifically modules implementig RedirFS filters. In the next part it considers different types of copression algorithms and the implementation specifics on the CryptoAPI framework in the linux kernel. Finally it describes the design and operation of the CompFLT module that implements a RedirFS compression filter. Benchmarks done using the CompFLT module are presented in the last section.

## Keywords

linux, kernel, module, compression, redirfs, file system

## Citace

Jan Podroužek: Compression filter for RedirFS framework, bakalářská práce, Brno, FIT VUT v Brně, 2007

# Compression filter for RedirFS framework

## Prohlášení

I hereby declare that this thesis is my own work and effort. Where other sources of information have been used, they have been acknowledged. This thesis was worked up under the supervision of Ing. Tomáš Kašpárek

.....

Jan Podroužek

May 14, 2007

© Jan Podroužek, 2007.

*Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.*

# Contents

<b>1</b>	<b>Prologue</b>	<b>3</b>
<b>2</b>	<b>Linux kernel module development</b>	<b>4</b>
2.1	Basics . . . . .	4
2.2	Compilation . . . . .	5
2.3	Debugging . . . . .	5
<b>3</b>	<b>RedirFS filter development</b>	<b>7</b>
3.1	Introduction . . . . .	7
3.2	Writing a filter . . . . .	7
<b>4</b>	<b>Compression</b>	<b>9</b>
4.1	Introduction . . . . .	9
4.2	Lossy compression algorithms . . . . .	9
4.3	Lossless compression algorithms . . . . .	9
4.3.1	Run-length encoding . . . . .	9
4.3.2	Dictionary coders . . . . .	10
4.3.3	Block-sorting compression . . . . .	10
4.3.4	Entropy encoding . . . . .	10
4.4	Lossless compression algorithm implementations . . . . .	11
4.4.1	deflate . . . . .	11
4.4.2	bzip2 . . . . .	11
4.4.3	lzo . . . . .	11
4.4.4	lzma . . . . .	11
<b>5</b>	<b>CryptoAPI</b>	<b>12</b>
5.1	Introduction . . . . .	12
5.2	Usage . . . . .	12
5.3	Expanding CryptoAPI . . . . .	13
5.4	Implemented cryptoapi modules . . . . .	14
5.4.1	null . . . . .	14
5.4.2	run-length encryption . . . . .	14
<b>6</b>	<b>Compflt</b>	<b>16</b>
6.1	Introduction . . . . .	16
6.2	Data structures . . . . .	16
6.2.1	File header . . . . .	16
6.2.2	Block . . . . .	18

6.3	Operations . . . . .	20
6.3.1	File read . . . . .	20
6.3.2	File write . . . . .	20
6.3.3	File open . . . . .	21
6.3.4	File release . . . . .	21
6.3.5	File llseek . . . . .	21
6.4	Block placement logic . . . . .	21
6.5	Command line parameters . . . . .	21
6.6	User-space interaction . . . . .	22
6.6.1	compft_ctl . . . . .	22
6.6.2	compft_stat . . . . .	22
<b>7</b>	<b>Benchmarks</b>	<b>24</b>
7.1	Tests and test data . . . . .	24
7.2	Results . . . . .	24
7.3	Conclusion . . . . .	24
<b>8</b>	<b>Conclusion</b>	<b>26</b>
8.1	Future additions . . . . .	26
<b>A</b>	<b>Graphs</b>	<b>28</b>

# Chapter 1

## Prologue

The aim and main goal of this bachelor thesis is the design and implementation of a linux kernel module, implementing a compression filter for the RedirFS [9] (Redirecting FileSystem) framework developed by *František Hrbata*.

The first part is an introduction to linux kernel module development. It covers the problematic of compilation, specific features of development in the kernel and debugging. The following section continues by explaining the way the RedirFS API is used to implement a filter.

Chapter four considers different types of compression algorithms and specific compression algorithms and programs that use them. This is complemented by next chapter which explains implementation specifics of the linux kernel CryptoAPI framework.

The main part is the description of the design and operation of the CompFLT kernel module which implements a compression filter using CryptoAPI as its compression backend. Benchmarks done using this filter are shown in chapter seven.

## Chapter 2

# Linux kernel module development

### 2.1 Basics

Linux kernel modules are pieces of kernel code which are loaded and unloaded on demand (provided they are not used at that moment). They can provide new functionality without the need to rebuild the kernel or even restart the computer.

Every module needs to contain two basic functions. The first one `init_module` is called when the module is loaded into the kernel and its purpose is to initialize everything the module needs to initialize. The second one `cleanup_module` is called just before the module is unloaded from the kernel and should provide a safe way to unload the module. Their prototypes can be seen in figure 2.1.

```
#include <linux/kernel.h>
#include <linux/modules.h>

int __init init_module(void);
void __exit cleanup_module(void);
```

Figure 2.1: `init_module` and `cleanup_module` function prototypes

The `__init` macro causes the function and its data to be unloaded from memory after it is finished if the code was compiled built-in and not as a module. Analogously the `__exit` macro causes the `cleanup_module` function to be omitted when the code is compiled into the kernel.

As of version 2.4 of linux it is possible to give arbitrary names to these two functions and register them using the `module_init` and `module_exit` macros (figure 2.2). This is mandatory from version 2.6.

```
module_init(my_init_function);
module_exit(my_exit_function);
```

Figure 2.2: Module init and cleanup functions registration

In order to identify and describe a module the `MODULE_AUTHOR`, `MODULE_DESCRIPTION` and `MODULE_LICENSE` macros are provided. Their use can be seen in figure 2.1. The value

set in the `MODULE_LICENSE` macro is also used to determine if the kernel is tainted, that is if it contains any closed-source code.

```
MODULE_LICENSE('Dual MIT/GPL');
MODULE_AUTHOR('Jan Podrouzek <xpodro01@stud.fit.vutbr.cz>');
MODULE_DESCRIPTION('Compression filter for the RedirFS Framework');
```

There are a few things to keep in mind while developing any kernel code. Code inside the kernel has access only to symbols inside the kernel itself, or to those exported by other modules. In order to make a symbol available outside a module, it has to be explicitly exported using the `EXPORT_SYMBOL` macro. Another important thing to keep in mind is that the kernel is heavily concurrent environment even on single processor computers (preemption). This leads to the need to use atomic data types, spin locks, semaphores (specially mutexes) and other mechanism to prevent inconsistencies or race conditions.

## 2.2 Compilation

The easiest way to compile kernel modules outside the main kernel tree is to use the Kbuild build mechanism. It requires a configured kernel with the full source code. Figure 2.3 shows a sample Makefile and Kbuild files that build a module named `compflt.ko` composed of 3 source files.

```
[Makefile:]
all:
    make -C /lib/modules/$(uname -r)/build M=$(PWD)

[Kbuild:]
obj-m := compflt.o
compflt-y := part1.o part2.o part3.o
```

Figure 2.3: sample Makefile and Kbuild files

## 2.3 Debugging

The best, simplest and recommended debugging method is with debugging output at key points in the code. This is achieved using the `printk`[10] function which has the same functionality as the `printf` functions known from the standard C library[16]. In addition `printk` accepts an integer value defining the severity of the message. This value ranges from `KERN_DEBUG` to `KERN_EMERG`. If no level is specified it defaults to `DEFAULT_MESSAGE_LOGLEVEL` defined in *kernel/printk.c*.

If this simple approach of debugging output does not work an interactive debugger such as `gdb`[8] can be used. An in-kernel debugger can be also used, such as *kdb* [12] (which is included in the official linux kernel tree) or *kgdb* [13].

To prevent data and hardware loss kernel code should be developed and tested either on a different machine connected using a serial console, or in a virtual machine. Some freely available open source virtual machines are *qemu* [1] or *User Mode Linux* (uml). Uml is

a linux virtual machine implemented as a separate architecture in the kernel itself , and is a part of the main linux tree as of version 2.6.9. Both qemu and uml use a regular file for their root filesystem.

## Chapter 3

# RedirFS filter development

### 3.1 Introduction

RedirFS [9, 17] replaces native Virtual filesystem switch (VFS) operations of files, inodes and dentries. Specific filters that register with RedirFS provide functions running before and/or after the original operations. The functions can in effect extend, modify or completely replace the behaviour of the original operations.

Individual filters can be attached to a directory, subtree or a single file, presenting away to filter only specific parts of the filesystem. In case of multiple filters attached to the same object a priority (set in each filter) determines the order in which they are called. Filters can dynamically change registered filters and attached filesystem subtrees.

### 3.2 Writing a filter

A RedirFS filter module needs to include the *redirfs/redirfs.h* header file. Different filters are identified by a handler of type `struct rfs_filter`, which is registered using the `rfs_register_filter` function as can be seen in figure 3.1.

```
rfs_filter compflt;  
struct rfs_filter_info flt_info = {“compflt”, 999, 0};  
  
compflt = rfs_register_filter(&compflt, &flt_info);
```

Figure 3.1: registering a filter with redirfs

`struct rfs_filter_info` defines the name of the filter, its priority and flags. The priority field serves as a way to determine calling order between multiple filters (higher priority filter being called first). The flags field is not used at present.

The second step is to tell RedirFS which operations will the filter intercept and which functions implement the pre and post calls. This is done by first filling an array of `struct rfs_op_info`, each containing a constant defining which operation is to be filtered and what type of object it applies to (e.g. open operation on a regular file) and function pointers to the pre and post calls (if only one is desired then the second can be set to NULL). This array is then passed to the `rfs_set_operations` function. The array has to be terminated properly with an item containing `RFS_OP_END` as its operation identifier. A sample operation registration can be seen in figure 3.2

```

static struct rfs_op_info ops_info[] = {
    {RFS_REG_FOP_OPEN, f_pre_open, NULL},
    {RFS_REG_FOP_RELEASE, NULL, f_post_release},
    {RFS_REG_FOP_READ, f_pre_read, NULL},
    {RFS_REG_FOP_WRITE, f_pre_write, NULL},
    {RFS_OP_END, NULL, NULL}
};
rfs_set_operations(compflt, ops_info);

```

Figure 3.2: registering operations with redirfs

The next step is to attach the filter to a directory or a subtree. Every path is defined in a single `struct rfs_path_info`. The first field specifies the path and the second defines the type of inclusion (only the directory itself, the whole subtree, etc.). Every path is then registered by passing it to the `redirfs_include_path` function as seen in figure 3.3

```

struct rfs_path_info pinfo = {"/tmp", RFS_PATH_INCLUDE|RFS_PATH_SUBTREE};
rfs_set_path(compflt, &pinfo);

```

Figure 3.3: registering a path with redirfs

Filters are by default inactive and have to be activated with the `redirfs_activate_filter` function.

# Chapter 4

## Compression

### 4.1 Introduction

Data compression is a process of encoding input data into another data in fewer bits (most of the times). There are two main types of data compression algorithms:

- lossless compression
- lossy compression

### 4.2 Lossy compression algorithms

Lossy compression is a method of compressing that drops some of the original data. Lossy compression methods achieve better compression ratios than lossless ones, which is balanced by the fact that the exact original data can't be retrieved from it.

Lossy compression is mostly used to compress multimedia files such as video and audio, where a certain amount of dropped data does not affect the resulting reproduction too much. This is mostly allowed by the limited properties of the human sensory system.

In the cases of our compression module the lossy compression method is not a viable option because the data being compressed is arbitrary in nature.

### 4.3 Lossless compression algorithms

Lossless compression employ the fact that almost all data contains some level of redundancy. All the lossless compression algorithms are reversible without any loss of data (hence the name).

There are many types of algorithms that allow to use the redundancy within any data to make the resulting data smaller. Most of the times these compression algorithms are used in conjunction to achieve the best compression results. The next few sections will describe some of the main lossless compression algorithms available.

#### 4.3.1 Run-length encoding

Run-length encoding [5] is an algorithm that encodes consecutions of same data by saving the count of repetitions and one occurrence of the data. This algorithm is very effective on

data that consists of long lines of same data. It is also used as part of other compression algorithms.

Some variations of run-length encoding are used in image data compression [15] and although this algorithm is lossless in nature there are variants of lossy run-length algorithms used in image compression.

### 4.3.2 Dictionary coders

Dictionary compression algorithms substitute parts of the data by references to data contained in a dictionary. There are two main types of dictionaries differing in the way they construct the dictionary.

- static
- dynamic

Static dictionaries contain preset data entries, that do not change in the process of compression. Dynamic dictionaries mostly start with a static set of entries, which change in the process of compression depending on the content of the compressed data.

Dictionary compression algorithms include:

- LZ77
- LZ78
- LZW

### 4.3.3 Block-sorting compression

Block-sorting [2] compression algorithms transform the input data in a way that the result contains patterns which are easily compressed by other compression algorithms such as run-length encoding or Huffman coding. An example of block-sorting is the Burrows-Wheeler transform. The important thing is that the transform has to be reversible in order to be incorporated in compression of data.

### 4.3.4 Entropy encoding

Entropy encoding substitutes pieces of data with codes of variable length. Assigning the shortest codes to symbols which are contained the most in the input data. These types of algorithms need some statistical data on the input data in order to decide which symbols get substituted to specific codes. Commonly used entropy encoding algorithms are:

- Huffman coding
- Adaptive Huffman coding
- Arithmetic coding

## 4.4 Lossless compression algorithm implementations

### 4.4.1 deflate

The Deflate [4, 7] compression algorithm is the combination of Huffman coding and LZ77 compression. There are many implementations of this algorithm including *pkzip* and *zlib*. This compression algorithm is also used in the widely used *gzip* compression utility.

### 4.4.2 bzip2

Bzip2 is a highly efficient and relatively fast compression algorithm. It can achieve compression ratios close to the best available compression algorithms while being faster both in compressing and decompressing the data.

Bzip2 comprises of a combination of many compression algorithms, mainly the Burrows-Wheeler block sorting algorithm, and Huffman coding. The best known implementation is the *libbzip2* library [20] and the *bzip2* compression utility.

### 4.4.3 lzo

The lzo compression algorithm [11] is a form of block compression algorithm derived from the *Lempel-Ziv* algorithm designed with speed as the main concern, favouring decompression speed over compression speed.

### 4.4.4 lzma

Lempel-Ziv-Markov algorithm [14] is a compression algorithm developed for the *7z* [18] achieving a very high compression ratio.

# Chapter 5

## CryptoAPI

### 5.1 Introduction

CryptoAPI is a framework in the linux kernel used for data transformation. It is designed in three layers

- algorithm API - provides an interface for registering new algorithms
- transform ops - implementation of the transform algorithms
- transform API - user(developer) interface

There are three types of transforms in CryptoAPI:

- cipher
- digest
- compress

Our focus is on the compress transform, but most of the logic is common to all the transform types. At current the main linux kernel tree contains only one compression transform implementing the `deflate` algorithm using the linux zlib library.

### 5.2 Usage

An example on how to use this API can be seen in figure 5.1. First a handle is registered using the allocation function specific to the type of the requested transform. In the case of a compression transform this is `crypto_alloc_comp`. The first parameter defines the requested algorithm and the second and third are used for miscellaneous flags. After the handle gets registered, it can be used in transform functions such as `crypto_comp_compress` (every transform type has its own specific transform functions). After the handle is no longer needed, it has to be freed by the `crypto_free_tfm` functions.

There are also some utility functions available, such as `crypto_has_alg` (and its transform specific counterparts) which is used to verify the availability of a specific transform algorithm.

```

struct crypto_tfm *tfm;
tfm = crypto_alloc_comp('rle', 0, 0);
if (IS_ERR(tfm))
    fail();
crypto_comp_compress(tfm, in, slen, out, &dlen);
...
crypto_comp_decompress(tfm, in, slen, out, &dlen);
crypto_free_tfm(tfm);

```

Figure 5.1: Cryptoapi compression transform usage example

### 5.3 Expanding CryptoAPI

The essential component in registering a new transform with CryptoAPI is the `crypto_alg` structure:

```

struct crypto_alg {
    struct list_head cra_list;
    u32 cra_flags;
    unsigned int cra_blocksize;
    unsigned int cra_ctxsize;
    unsigned int cra_alignmask;
    int cra_priority;
    char cra_name[CRYPTO_MAX_ALG_NAME];
    char cra_driver_name[CRYPTO_MAX_ALG_NAME];

    int (*cra_init)(struct crypto_tfm *tfm);
    void (*cra_exit)(struct crypto_tfm *tfm);

    union {
        struct cipher_alg cipher;
        struct digest_alg digest;
        struct compress_alg compress;
    } cra_u;

    struct module *cra_module;
};

```

#### **cra\_list**

A linked list containing all currently registered transforms.

#### **cra\_name**

Unique identifier of the transform within cryptoapi. Used when requesting the transform (as can be seen in the previous section).

#### **cra\_flags**

Specifies the type of the transform, possible values are `CRYPTO_ALG_TYPE_CRYPTO`, `CRYPTO_ALG_TYPE_CIPHER` and `CRYPTO_ALG_TYPE_DIGEST`.

### **cra\_init**

Function called when the transform is initialized.

### **cra\_exit**

Cleanup function called when the transform is being deinitialized.

### **cra\_u**

Union of structures containing information specific to the type of the transform. In the case of a compression transform it contains pointers to functions used to compress and decompress data, as can be seen in figure 5.2

```
struct compress_alg {
    int (*coa_compress)(struct crypto_tfm *tfm, const u8 *src,
                       unsigned int slen, u8 *dst, unsigned int *dlen);
    int (*coa_decompress)(struct crypto_tfm *tfm, const u8 *src,
                          unsigned int slen, u8 *dst, unsigned int *dlen);
};
```

Figure 5.2: compression transform specific functions

The registration and unregistration of a transform is done using the `crypto_register_alg` and `crypto_unregister_alg` functions. Their function prototypes are shown in figure 5.3.

```
int crypto_register_alg(struct crypto_alg *alg);
int crypto_unregister_alg(struct crypto_alg *alg);
```

Figure 5.3: cryptoapi (un)register function prototypes

## **5.4 Implemented cryptoapi modules**

### **5.4.1 null**

In order to test the `compft` module independently of any compression algorithm a null cryptoapi module was written. This module copies its input data to the output without changing it. It is implemented in a single file `crypto/null.c`.

### **5.4.2 run-length encryption**

This is a cryptoapi module implementing a simple run-length encryption algorithm as described in [3]. The main reason to implement this algorithm in particular was to complement the existing `deflate` compression algorithm which achieves good compression ratios but is relatively slow. A sample of uncompressed and rle-compressed data can be seen in figure 5.4. The whole module is implemented in a single file `crypto/rle.c`.

```
input:
0x61 0x61 0x61 0x61 0x61 0x62 0x62 0x62
0x63 0x64 0x61 0x61 0x61 0x61 0x61 0x62
compressed:
0x61 0x61 0x03 0x62 0x62 0x01 0x63 0x64
0x61 0x61 0x03 0x62 0x62 0x01 0x63 0x64
```

Figure 5.4: sample rle data

# Chapter 6

## Compflt

### 6.1 Introduction

Compflt is the kernel module that implements filters for file and inode operations needed to compress/uncompress data within a file on-the-fly. It is independent of any specific compression algorithm by using the CryptoAPI framework to do the actual compression work.

### 6.2 Data structures

#### 6.2.1 File header

The file header written to the compressed file contains three items. First a magic number [19], which is a reasonably long unique sequence of bytes written at the start of the file in order to recognise it as one compressed by Compflt. The sequence of bytes chosen is a 4-byte representation of the authors birth date.

```
0x06 0x10 0x19 0x82
```

Its uniqueness was verified by checking the magic file (to be found at `/usr/share/file/magic` on most linux installations) in a recent version of a Linux distribution. A corresponding entry in a magic file would be:

```
0 long 0x06101982 Compflt compressed file
```

Second is the compression method identifier. This is used to determine which compression algorithm was used to compress the file. It is 1-byte long and corresponds to the `method` field in `struct fheader`. Last is the block size used to compressed data blocks within the file. The layout of the file header can be seen in figure 6.1.



Figure 6.1: File header layout

```

struct cflt_file {
    struct list_head all;
    struct list_head blks;
    struct inode *inode;
    unsigned int method;
    unsigned int blksize;
    unsigned int size_u;
    atomic_t compressed;
    atomic_t dirty;
    atomic_t cnt;
    wait_queue_head_t ref_w;
    spinlock_t lock;
};

```

Figure 6.2: struct fheader

As a representation of the file header and file-specific attributes in the code `struct file` is used. Its definition can be seen in figure 6.2. The fields have the following meaning:

**all**

This field links all `cflt_file` structures together in one list. This list starts at `cflt_file_list` defined in `file.c`

**blks**

This field is the head for a list of all blocks contained in this file. They are linked using the `file` member of `struct cflt_block`.

**inode**

Holds a pointer to the inode to which this `cflt_file` corresponds.

**method**

This field holds the number of the compression method used in the file. It is used as an index into the `comp_methods` array defined in `compression.c`:

```
char *comp_methods[]={‘‘deflate’’, ‘‘lzf’’, ‘‘bzip2’’, ‘‘rle’’, NULL};
```

For example a file compressed with the deflate compression method would have the field set to 0. This is also one of the two fields within `struct cflt_file` that is written to the compressed file.

**blksize**

Defines the block size of blocks in the file. Its value boundaries are defined by the `CFLT_BLKSIZE_MIN` and `CFLT_BLKSIZE_MAX` constants. This field is written to the file.

**size\_u**

Represents the total uncompressed size of the file. At this moment used only to set appropriate offset values when the `lseek` syscall is used with the `whence` values set to `SEEK_END`.

**compressed**

A flag identifying the file as being compressed using Compflt.

**dirty**

A flag that is set whenever the state of any field from `struct cflt_file` which is written to the file has changed. It is unset when the state is written to the file.

**cnt**

Reference count.

**ref\_w**

Wait queue used when the file is getting deinitialized, to ensure there are no references to it.

**lock**

A spin lock used to protect non-atomic members of the structure.

In order to minimize read/write operation upon the compressed file a file header cache is in place. It is implemented using the kernel slab allocator. The name of the cache is defined in the `CACHE_NAME` constant defined in *file.c*.

**6.2.2 Block**

The block has a header that contains a mapping between the position and size of the compressed data block saved in the compressed file and the position and size of the same data block in its uncompressed form. The mapping is necessary for both read and write operations and is used to identify blocks that are to be part of the current operation.

An early version of Compflt contained all the block headers in a reserved space at the beginning of the file just behind the file header. This demanded a reserved space for these headers which represented a constant size overhead. For example to be able to support a file size of 2 GB using 4 kB data blocks, the total overhead would be around 6 MB. Even if a block size as high as 16 kB would be used, the resulting overhead would be approximately 1.5 MB. The advantage of this approach was the ability to read all the block headers from a contiguous area within the compressed file. This method was later modified so that every block header was saved in the compressed file just before the data as can be seen in figure 6.3

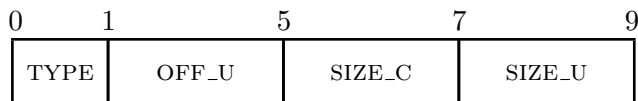


Figure 6.3: Block header layout

This approach removed the constant size overhead of the previous approach, although it degraded performance slightly because the headers are read from the full length of the file. This disadvantage is partially compensated by the use of a block cache. The structure representing a block is `struct block` and can be seen in figure 6.4. The fields have the following meaning:

```

struct cflt_block {
struct list_head file;
    unsigned int type;
    unsigned int off_u;
    unsigned int off_c;
    unsigned int size_u;
    unsigned int size_c;
    struct cflt_file *par;
    atomic_t dirty;
    char *data_u;
    char *data_c;
};

```

Figure 6.4: struct block

#### **file**

This field links all blocks that belong to a single file in a list. The list starts at the `blks` member of the corresponding `struct file`.

#### **type**

Identifies the blocks type. This is to support multiple block header types and also free blocks (blocks not currently holding any data). At the moment the possible values for this field are `CFLT_BLK_NORM` and `CFLT_BLK_FREE`.

#### **off\_u**

The offset from the start of the uncompressed file to the start of the uncompressed data in this block. This field is written to the compressed file in 4 bytes.

#### **off\_c**

The offset from the start of the compressed file to the beginning of this block in it. This field is not written to the file, because it is implied by the blocks position at the time when it is read from the file.

#### **size\_u**

The size of the uncompressed data in this block. This field is written to the compressed file in 2 bytes, limiting the block size to 65535 bytes.

#### **size\_c**

The size of the compressed data in this block. This field is written to the compressed file in 2 bytes, which limits the largest block size to be 65535 bytes.

#### **par**

A pointer to the `cflt_file` in which this file is contained.

#### **dirty**

A flag set when any member of this structure that is written to file gets modified. These are `off_u`, `size_u` and `size_c`. It is unset when the block is updated in the file. This is implemented in `block_write_header`.

## **data\_u**

This is a temporary pointer to the uncompressed data from the block. It is present purely for convenience.

## **data\_c**

This is a temporary pointer to the compressed data from the block. It is present purely for convenience.

The cache of blocks (only the header data is actually saved in the structure) is implemented using the kernel slab allocator. The name of the cache is defined in the `CACHE_NAME` constant defined in *block.c*.

## **6.3 Operations**

### **6.3.1 File read**

We register a precall function for the file read operation. The function is named `f_pre_read`. In the first place it tries to get a file header corresponding to the file upon which the read operation was called. This is done in the `cflt_file_get` function which either returns a file header from the cache, or it initializes a new file header and tries to populate it by reading the compressed file. If the read fails or the file is not recognised as a compressed one, the read pre\_call function returns operation to the original read operation.

If the file header was successfully retrieved, the requested data is saved to the user buffer through a call to `cflt_read` function. This function iterates over the list of blocks in the file. If a block contains some part of the requested data, the compressed data is read from the file, uncompressed and the appropriate part of the uncompressed data is copied over to the user buffer.

The last task is to set the return value to the size of the data saved in the user buffer, and increment the current position within the file by the same amount.

### **6.3.2 File write**

We register a precall function for the file write operation. The function is named `f_pre_write`. Identically to `f_pre_read` the first thing done is the retrieval of a file header. After that an additional check is done to determine if the `O_APPEND` flag is set for the file. If it is set the position argument is set to the uncompressed file size (this value is stored in the `size_u` field of `struct file`).

After that the requested write operation is performed by calling the `cflt_write` function. It first iterates over all existing blocks. If a block is matched (meaning it contains some part of the file that we want to write to), it is read from the compressed file and decompressed. Next the appropriate part of the input buffer gets written to the corresponding part of the uncompressed data block. The block is then recompressed and written back to the file. If there is some data left to be written after the iteration is finished, the function creates new data blocks, compresses them and writes them to the file until there is no data left in the input buffer.

Same as in `f_pre_read` the last action is to update the file position and set the return value to the count of bytes written.

### 6.3.3 File open

We register a precall function for the file open operation. The function is named `f_pre_open`. The sole purpose of this function at this moment is to detect the `O_TRUNC` file flag. If this flag is set the function invalidates all blocks within the specified file, updates its size to zero as well as set the compressed flag to zero.

### 6.3.4 File release

A post call function is registered for the file release operation. The function is named `f_post_release`. This function is in charge of writing all file headers marked as dirty to the compressed file. This in effect also writes all block headers marked as dirty.

### 6.3.5 File llseek

The precall for the `llseek` file operation is used to update the `i_size` value of the corresponding `vfs` inode in order for the `lseek` system call to work correctly on compressed files.

## 6.4 Block placement logic

Every new and changed block is written to the file in a place determined by the `cflt_file_place_new_block` and `cflt_file_place_old_block` functions respectively.

In the case of a new block `compflt` first tries to find an existing free block which has enough space to accommodate it. If such a block is found the new one is written in its place. If there is space left over (the new block is smaller than the existing free block), a new free block is placed in the free space. If there are no suitable free blocks available the new block is placed at the end of the file.

Old blocks have a more complex placement algorithm. If the size of the modified block is unchanged, or if it is the last block in the file, it is left in its old place. In the case the block has shrunk (smaller compressed size) and we cannot fit a new free block in the freed space the block is moved (explained in the next paragraph) to a different place. If the block has grown (bigger compressed size) and the next block is a free one the block expands over the space currently occupied by the free block. If that is not possible the block has to be moved.

If the modified block has to be moved `compflt` first checks if the previous block is a free one. If so, the modified block is shifted into the free space (reducing the size of the free block). If that is not possible the algorithm tries to find a suitable free block anywhere within the file. If that fails too the block is moved to the end of the file.

## 6.5 Command line parameters

The *Compflt* module accepts two optional parameters on load. First is *in\_method* which defines the compression method that will be used. If the requested method is unknown or the parameter is not set, the default one (*deflate*) is used. The second parameter is *in\_blksize* and it permits the user to set the initial block size.

## 6.6 User-space interaction

### 6.6.1 compflt\_ctl

Compflt possesses the ability to query and change the compression method and block size at any time using two control files, `/proc/fs/compflt/method` and `/proc/fs/compflt/blksize`. Both files when read return their current value.

If the `method` file is written to, compflt compares the string to all known compression methods and if it finds a match it verifies that that compression method is available by calling `crypto_alg_available` (if the algorithm isn't found within the ones currently registered this function tries to load a kernel module of the same name eliminating the need to load the compression modules beforehand).

If the `blksize` file is written to, the passed values is compared to the boundaries defined in `compflt/compflt.h`. If the value is within the acceptable limits the block size is changed.

In order to make querying and setting of these values easier a shell script control utility `compflt/util/compflt_ctl` is provided along with a manual page. A sample usage session can be seen in figure 6.5

```
defiant-qemu:~# ./compflt_ctl
method=lzf
bsize=4096
defiant-qemu:~# ./compflt_ctl bsize 2048
compflt: block size set to 2048 bytes
bsize=2048
defiant-qemu:~# ./compflt_ctl method lzf
compflt: compression method set to 'lzf'
method=lzf
```

Figure 6.5: compflt\_ctl script usage

### 6.6.2 compflt\_stat

Compflt also provides a way to query per-inode compression statistics by reading the `/proc/fs/compflt/stat` file. A sample output can be seen in figure 6.6. There is also an accompanying shell script `compflt/util/compflt_stat` that presents the data in a more human-readable way. An example of its output can be seen in figure 6.7

```
defiant-qemu:~# cat /proc/fs/compflt/stat
ino      alg      blksize  comp      ohead     decomp
1120431  deflate  4096     4559      45         15501
1120440  deflate  4096     2229      27         5782
1120441  deflate  4096     2334      27         6522
1120450  deflate  4096     349       18         635
1120453  lzf     4096     7425      45         15501
```

Figure 6.6: Example statistics output

```
defiant-qemu:~# ./compflt_stat
==== per-inode stats ===
<1120431>      deflate 15501 -> 4604 (70.2987%)
<1120440>      deflate 5782 -> 2256 (60.9824%)
<1120441>      deflate 6522 -> 2361 (63.7994%)
<1120450>      deflate 635 -> 367 (42.2047%)
<1120453>      lzf      15501 -> 7470 (51.8096%)

==== per-alg stats ===
<deflate>      28440 -> 9588 (66.2869%)
<lzf>          15501 -> 7470 (51.8096%)

avg ratio achieved: 61.1798%
```

Figure 6.7: Example compflt\_stat output

# Chapter 7

## Benchmarks

### 7.1 Tests and test data

As a suitable and tested data collection the *Canterbury corpus* and *Large corpus* from the Canterbury corpus [6] benchmark suite were chosen. It consists of an assortment of files such as a English text, a html source, c source, excel spreadsheet, executable file and more.

The tests were done using an automated bash script which copies the two corpuses to a directory to which the compfft filter is attached and then it copies them to a directory without the filter attached. Both operations are timed using the standard built-in *time* function. The total size of the compressed corpuses is also calculated. The script iterates over different compression algorithms and different block size settings. Every test is repeated 3 times and their values averaged.

The machine used to run these tests consists of a Pentium M 1500 MHz processor, 768 MB of RAM memory and is running the 2.6.21 version of the Linux kernel.

### 7.2 Results

The tables 7.1 and 7.2 show the averaged results for the canterbury and large corpuses respectively. The *cp* method in these tables represent results done while the compfft filter was not loaded.

Some graphs that might be more exemplificative can be found in appendix A.

### 7.3 Conclusion

The deflate compression algorithm achieves the best compression ratios on all data types, but on the other hand has the worst compression times. Its use could be in situations where the data is not modified often and the resulting size is the main concern (for example backup data).

The lzf algorithm achieves a good and balanced combination of compression ratio and speed, making it the best choice for situations where the data is modified more often.

The rle compression module althou being relatively fast, has very bad compression abilities on 'average' data, making it unsuitable for most environments. The only possible use could be on specific data containing repetitions of characters, such as spaces.

Considering the block size used, the best setting in average is 4096 bytes. This can be explained by the setting of the disk device block size, and also by the usage of this size by

method	block size [ $B$ ]	ratio [%]	write [ $s$ ]	read [ $s$ ]
cp	n/a	100.00	0.0210	0.0097
deflate	2048	30.56	0.4187	0.1240
	4096	28.64	0.4213	0.1157
	8192	27.47	0.6233	0.1337
lzf	2048	54.98	0.0650	0.0347
	4096	53.79	0.0567	0.0320
	8192	54.98	0.0863	0.0470
rle	2048	90.56	0.0327	0.0453
	4096	90.32	0.0280	0.0477
	8192	90.20	0.0407	0.0480
null	2048	100.45	0.0327	0.0277
	4096	100.23	0.0280	0.0250
	8192	100.12	0.0407	0.0370

Table 7.1: Canterbury corpus test results

method	block size [ $B$ ]	ratio [%]	write [ $s$ ]	read [ $s$ ]
cp	n/a	100.00	0.0350	0.0343
deflate	2048	41.24	2.0833	0.7030
	4096	38.71	2.1967	0.5820
	8192	37.25	3.2970	0.6863
lzf	2048	81.83	0.5200	0.2558
	4096	80.18	0.3957	0.2053
	8192	79.06	0.5410	0.2700
rle	2048	106.17	0.3547	0.3093
	4096	105.95	0.2347	0.3010
	8192	105.84	0.3127	0.3537
null	2048	100.44	0.2890	0.2077
	4096	100.22	0.1673	0.1490
	8192	100.11	0.1943	0.1785

Table 7.2: Large corpus test results

user-space utilities such as *cp* for the buffers used in read / write operations. If a smaller block size is chosen more than one block is created during one write operation, if a larger block size is used the filter has to decompress the block before writing the subsequent data to it. In both cases operation speed suffers from these effects, and although a larger block size yields better compression ratios the benefit is small compared to the degradation of performance.

The overall best combination on the tested machine is the use of the *lzf* compression module and 4096 bytes long blocks.

# Chapter 8

## Conclusion

The main goal of this work was achieved and a working implementation of a RedirFS compression filter was developed. It has the ability to compress and decompress data on-the-fly, supports random access (both reading and writing) to the compressed file without the need to decompress it as a whole. An independency from any specific compression algorithm was achieved by the use of CryptoAPI as its compressoin backend. It also proved the usefulness of the RedirFS framework itself. Future development should make it an even better solution for data compression and an alternative to current methods of saving compressed data on computer storage devices. Some of the possible future extensions and additions are mentioned in the next section.

### 8.1 Future additions

- **mmap support**

Support for the mmap operation, which maps a files content to memory, making it easy to manipulate with standard memory functions. This is also needed to support execution of executable files from within a compressed directory.

- **directory compression utility**

An utility that compresses files that were created before a directory was included in the filters path settings. This utility could also automatically add the desired path to the compression filters paths.

- **defragmentation utility**

Because the compressed files can suffer from fragmentation, a utility that defragments these files could be usefull for situations where heavy file modifications ocured.

- **RedirFS configuration callback**

Support for the currently developed feature of the redirfs framework which lets filters register a special configuration callback that can pass various settings from user-space to filters in an unified way.

- **various optimizations**

Some optimizations could be made mainly to the write operations. Such cases as the overwriting of entire block (no need to decompress them), or limited use of a block cache in sequential write operations.

- **more compression modules**

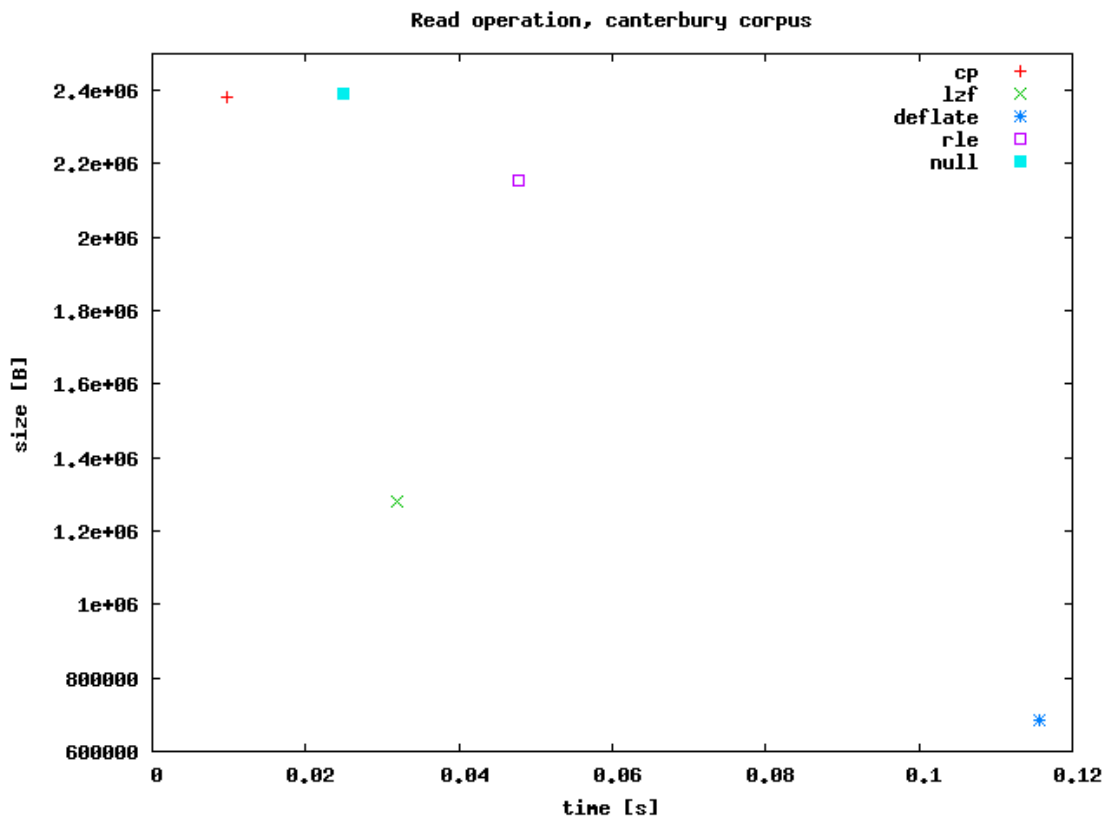
Further development of new compression modules with different compression algorithms suitable for various specific environments. A bzip2 compression module is currently under development.

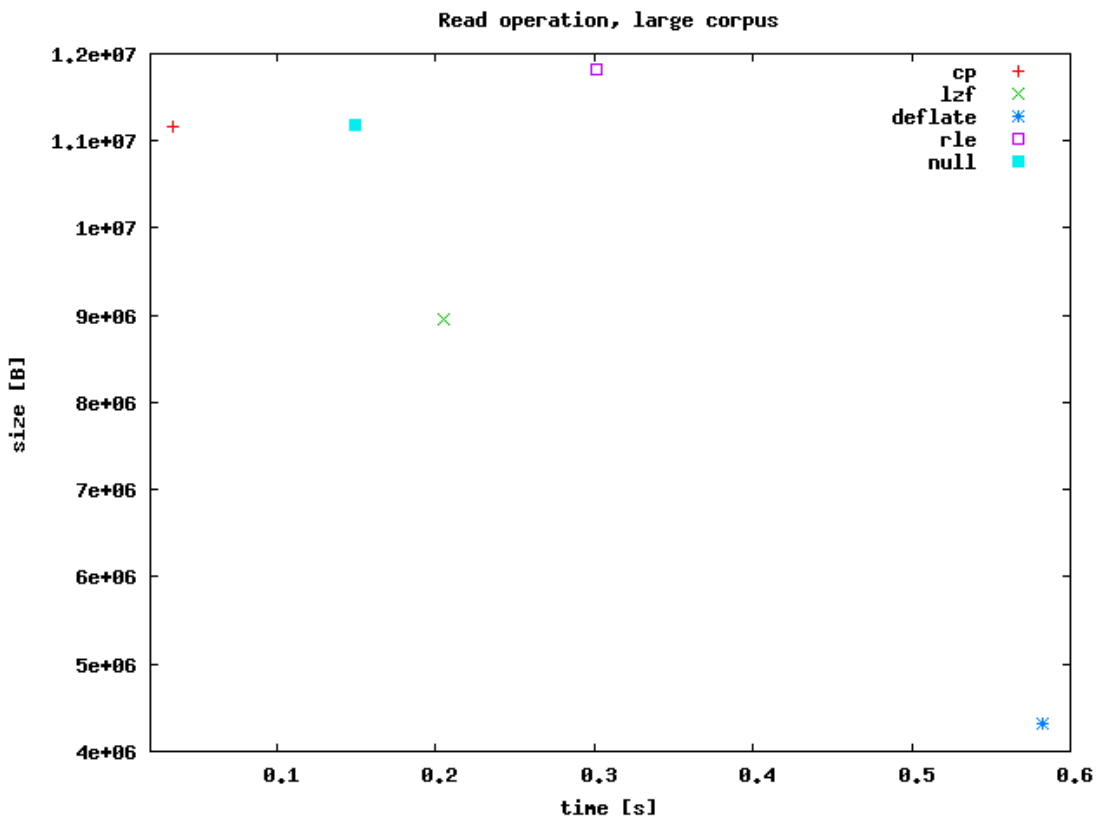
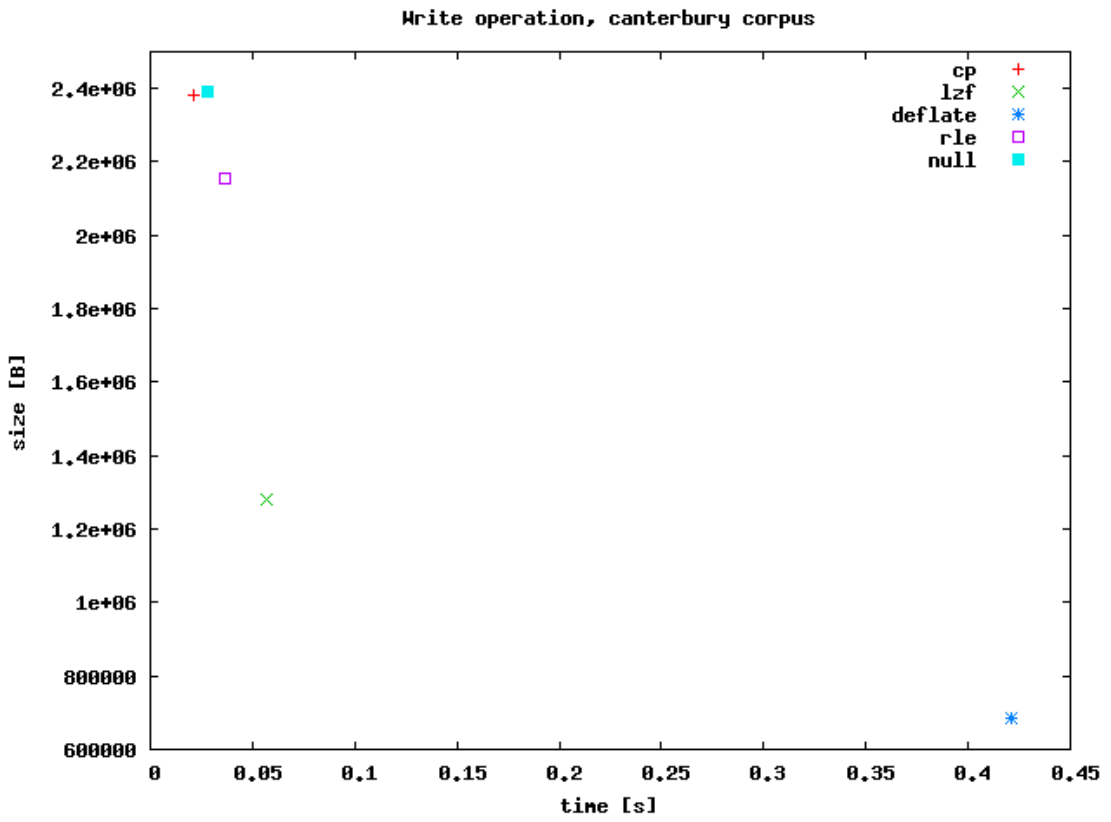
- **configs support**

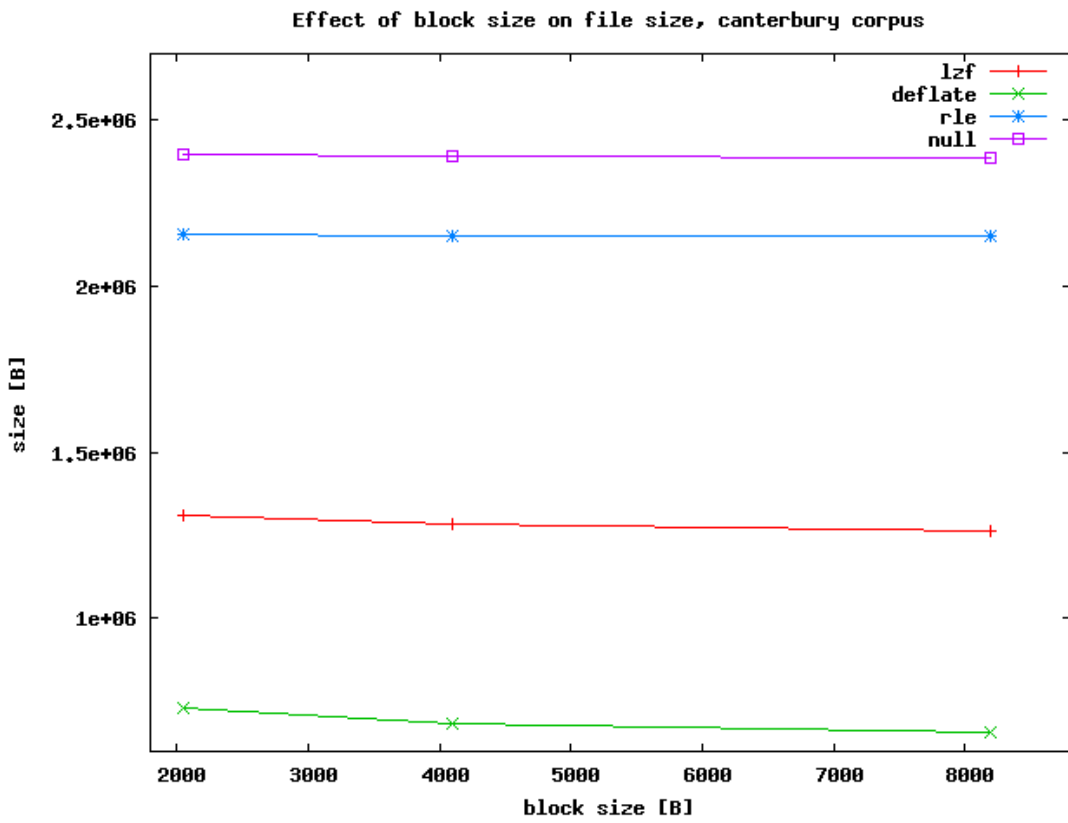
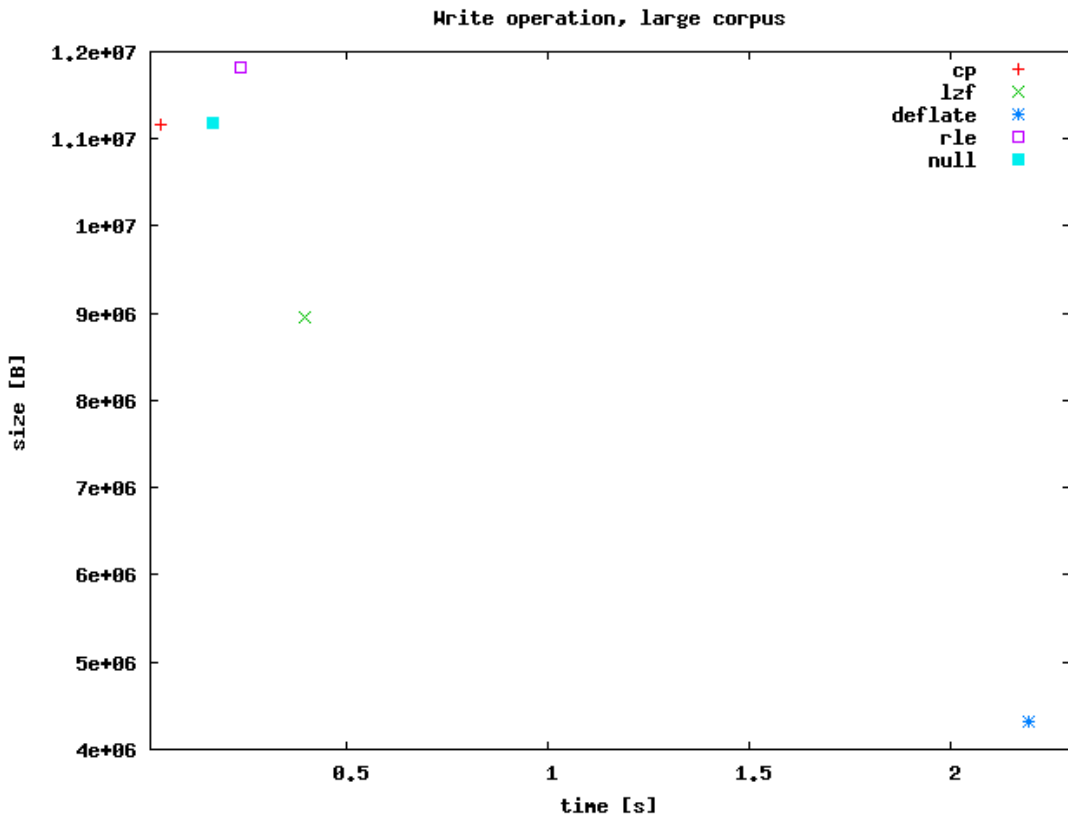
As the *procfs* file system, currently used to change compression parameters, was never ment to provide such functions and should only contain process specific information these functions should be migrated to a more suitable file system such as *configs*.

# Appendix A

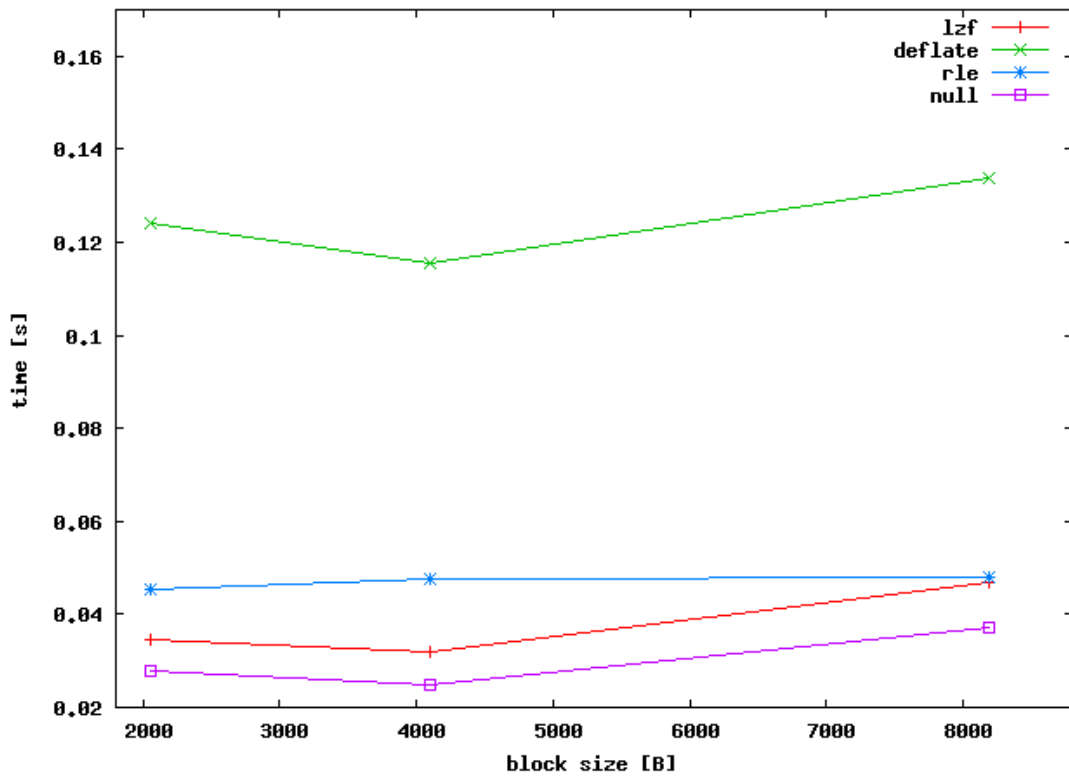
## Graphs



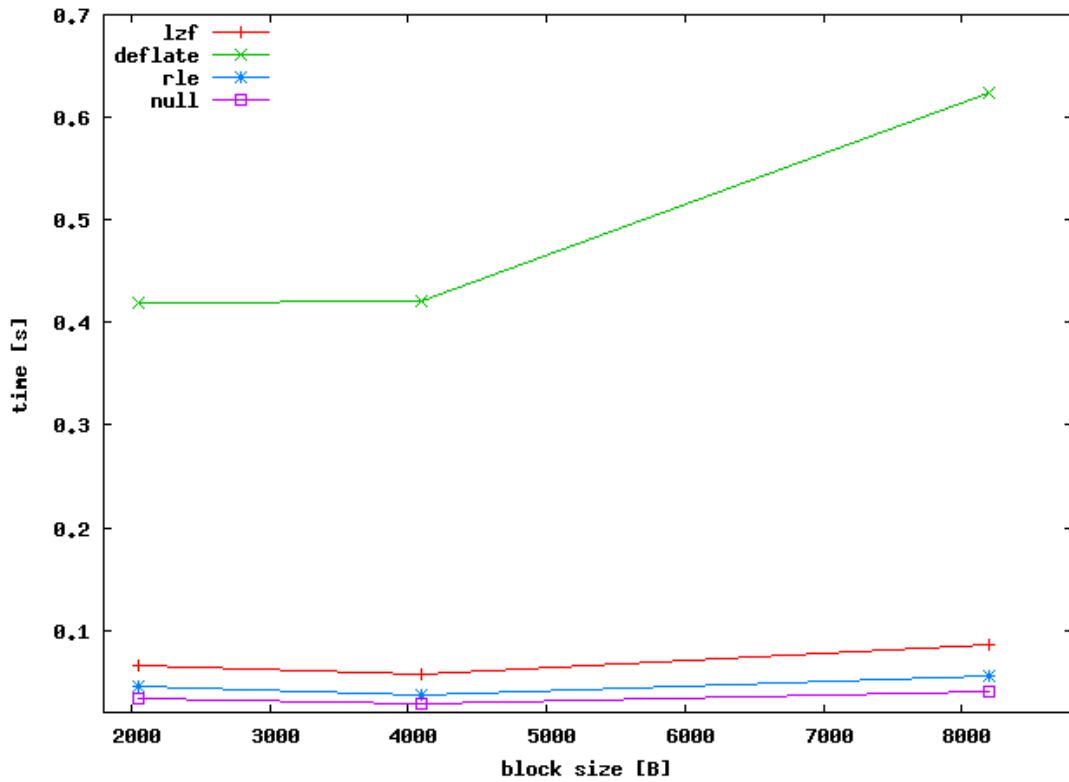




Effect of block size on read time, canterbury corpus



Effect of block size on write time, canterbury corpus



# Bibliography

- [1] Fabrice Bellard. Qemu - open source processor emulator.  
<http://fabrice.bellard.free.fr/qemu/>.
- [2] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, 1994.
- [3] Arturo San Emeterio Campos. Run length encoding.  
[http://www.arturocampos.com/ac\\_rle.html](http://www.arturocampos.com/ac_rle.html).
- [4] Peter Deutsch. Deflate compressed data format specification.  
<http://www.ietf.org/rfc/rfc1951.txt>.
- [5] Michael Dipperstein. Run length encoding discussion and implementation.  
<http://michael.dipperstein.com/rle/index.html>.
- [6] Tim Bell et al. The canterbury corpus. <http://corpus.canterbury.ac.nz/>.
- [7] Antaeus Feldspar. An explanation of the deflate algorithm.  
<http://zlib.net/feldspar.html>.
- [8] GNU. Gnu debugger. <http://www.gnu.org/software/gdb/>.
- [9] František Hrbata. Callback framework for vfs layer. Master's thesis, Brno University of Technology, Faculty of Information Technology, 2005.
- [10] Alesandro Rubini Jonathan Corbet and Greg Kroah-Hartman. *Linux Device Drivers, Third Edition*. O'Reilly Media, Inc., 2005.
- [11] Markus Franz Xaver Johannes Oberhumer. Lzo real-time data compression library.  
<http://www.oberhumer.com/opensource/lzo/>.
- [12] Web page. Kdb (built-in kernel debugger). <http://oss.sgi.com/projects/kdb/>.
- [13] Web page. Kgdb: Linux kernel source level debugger.  
<http://kgdb.linsyssoft.com/>.
- [14] Web page. Lempel-ziv-markov algorithm. <http://en.wikipedia.org/wiki/LZMA>.
- [15] Web page. Run-length encoding (rle).  
[http://www.fileformat.info/mirror/egff/ch09\\_03.htm](http://www.fileformat.info/mirror/egff/ch09_03.htm).
- [16] Wep page. The c standard library.  
<http://www.utas.edu.au/infosys/info/documentation/C/CStdLib.html>.

- [17] Web page. Redirfs. <http://www.redirfs.org>.
- [18] Igor Pavlov. 7-zip. <http://www.7-zip.org/>.
- [19] Eric S. Raymond. The magic numbers group.  
<http://www.catb.org/esr/magic-numbers/>.
- [20] Julian Seward. bzip2. <http://www.bzip.org/>.