Container Library and FUSE Container File System

Softwarepraktikum für Fortgeschrittene

Parallele und Verteilte Systeme
Institut für Informatik
Ruprecht-Karls-Universität Heidelberg

Michael Kuhn
Matrikelnummer: 2405219

2008-03-31

Contents

1. Introduction **3**
1.1. Requirements . . . 3
1.2. Goals . . . 3

2. Container Library **4**
2.1. License . . . 4
2.2. General Overhaul . . . 4
2.3. New Features . . . 4
2.3.1. Architecture Independence . . . 4
2.3.2. Thread-Safety . . . 4
2.3.3. Write Support . . . 4
2.3.4. File Hashing . . . 5
2.4. Application Programming Interface . . . 5

3. Container Tools **7**

4. FUSE Container File System **8**
4.1. Realization . . . 8
4.2. Implementation . . . 8

5. Evaluation **13**
5.1. Hardware . . . 13
5.2. Without Metadata . . . 13
5.3. With Metadata . . . 14

Appendices **16**

A. Usage Instructions **16**
A.1. Installing Required Packages . . . 16
A.2. Compiling Everything . . . 16
A.3. Generating `libct` Documentation . . . 16
A.4. FUSE file system . . . 16

B. Benchmark Program **17**

C. BSD License **20**

D. Benchmark Data **21**

Bibliography **22**

1. Introduction

There are cases when many small files must be stored in a file system. If these files are accessed frequently, metadata performance plays an important role. An attempt to decrease the metadata overhead is to maintain a reduced set of metadata for these files. This usually has to be done manually, because available file systems do not permit the user to change which metadata is stored. One approach is to pack them together in one file – a container – and thus have the file system only manage metadata for one file. Within this container, the files and their corresponding metadata can be managed arbitrarily. One advantage of such an approach is that the files' data and metadata are packed together and can be read efficiently – that is, if the container is not fragmented.

1.1. Requirements

It is obvious that the container format should enable random access to provide access times independent of the position of a file within the container. Existing formats either do not provide random access – like the `tar` format – or store too much metadata – like the `iso` format. Therefore a new container format was designed and implemented in [Hei07]. This existing implementation was used as a basis for all work presented in this report.

1.2. Goals

The goal of this practical was to maintain and enhance the container library and to create a FUSE[1]-based file system using it. The FUSE file system was planned to allow legacy applications to access containers via the POSIX API for file system access.

[1]Filesystem in Userspace – `http://fuse.sourceforge.net/`

2. Container Library

2.1. License

The container library was not licensed in any way, making future development and usage difficult. It has therefore been licensed under a 2-clause BSD license in agreement with the original author. A copy of the license text can be found in appendix C.

2.2. General Overhaul

The library was completely overhauled to provide consistently named functions and data types. Additionally, the comments within the code were modified to allow the automatic generation of an API documentation with Doxygen[1].

2.3. New Features

2.3.1. Architecture Independence

The original implementation did not honor the different sizes of data types on 32 and 64 bit architectures, thus making it impossible to use a container created with a 32 bit version of the library with a 64 bit version and vice versa. By using datatypes of a fixed size the containers can now be used on both 32 and 64 bit architectures without problems. Additionally, the size of the metadata structures used in the library is now independent of the architecture.

2.3.2. Thread-Safety

Because of the use of a shared file pointer and the functions `read` and `write` in combination with `lseek` the library was not thread-safe in its original form. Therefore the library was modified to use the `pread` and `pwrite` functions that do not modify the shared file pointer. It can now be used safely in multi-threaded applications.

2.3.3. Write Support

The original version of the container library lacked an easy-to-use method to create containers. Therefore, a convenient interface to add new files from either memory or an existing file was added to the library. For more information see `ct_file_create_*` and `ct_file_create_fast_*` in section 2.4.

[1] `http://www.stack.nl/~dimitri/doxygen/`

2.3.4. File Hashing

File data stored in the container is protected from silent corruption by storing a SHA-1 hash of it along with its metadata.

2.4. Application Programming Interface

`struct ct_container` This structure represents a container.

`struct ct_file` This structure represents a file within a container.

`struct ct_container* ct_container_open (char* path)` This function opens the container specified by the `path` argument and returns a `ct_container` pointer. This pointer can be used to perform further operations on the container.

`int ct_container_close (struct ct_container* container)` This function closes the container associated with the `container` argument and returns an error code.

`char* ct_container_read (struct ct_container* container, char* file_name)` This function reads the names of all files stored within the container associated with the `container` argument. The `file_name` argument must be `NULL` for the first call and a pointer returned by the function itself for subsequent calls.

`char* ct_container_read_offset (struct ct_container* container, uint32_t offset)` This function returns the name of the file at position `offset` within the container associated with the `container` argument.

`int ct_file_exists (struct ct_container*, const char* path)` This function checks whether the file with name `path` exists within the container associated with the `container` argument.

`struct ct_file* ct_file_open (struct ct_container* container, const char* path)` This function opens the file named `path` within the container associated with the `container` argument and returns a `ct_file` pointer. This pointer can be used to perform further operations on the file.

`struct ct_file* ct_file_open_offset (struct ct_container* container, uint32_t offset)` This function opens the file at position `offset` within the container associated with the `container` argument and returns a `ct_file` pointer. This pointer can be used to perform further operations on the file.

`const char* ct_file_name (struct ct_file* file)` This function returns the name of the file associated with the `file` argument.

`off_t ct_file_size (struct ct_file* file)` This function returns the size of the file associated with the `file` argument.

`char* ct_file_hash (struct ct_file* file, char* hash)` This function returns the SHA-1 hash of the file associated with the `file` argument and stores it in the buffer pointed to by the `hash` argument.

`ssize_t ct_file_read (struct ct_file* file, void* buffer, size_t count)` reads `count` bytes from the file associated with the `file` argument and stores it in the buffer pointed to by the `buffer` argument.

`int ct_file_close (struct ct_file* file)` closes the file associated with the `file` argument.

`off_t ct_file_seek (struct ct_file* file, off_t offset, int whence)` sets the file pointer of the file associated with the `file` argument according to the `offset` and `whence` arguments.

`struct ct_container* ct_container_create (char* path)` creates a file called `path` and returns a `ct_container` pointer. This pointer can be used to add files to the container with the following functions.

`int ct_file_create_fast_buffer (struct ct_container* container, const char* file_name, void* buffer, size_t size)` adds a new file called `file_name` to the container associated with the `container` argument. The contents of the file are read from the buffer pointed to by the `buffer` argument and is of size `size`.

`int ct_file_create_buffer (struct ct_container* container, const char* file_name, void* buffer, size_t size)` does the same as `ct_file_create_fast_buffer`, but sorts the array of files after each call. Otherwise this is done implicitly when `ct_container_close` is called.

`int ct_file_create_fast_path (struct ct_container* container, const char* file_name, const char* path)` adds a new file called `file_name` to the container associated with the `container` argument. The contents of the file are read from the file called `path`.

`int ct_file_create_path (struct ct_container* container, const char* file_name, const char* path)` does the same as `ct_file_create_fast_path`, but sorts the array of files after each call. Otherwise this is done implicitly when `ct_container_close` is called.

For more information, also see the Doxygen documentation, which includes code examples and more.

3. Container Tools

To provide a possibility to work with containers from the command line, several command line tools were implemented. They provide basic functionality like their POSIX counterparts `cat`, `cp` and `ls`.

ctmk

This tool creates a new container with all files in a given directory.
Example:

```
$ ctmk ~/etc.ct /etc
```

ctcat

This tool prints the contents of a container file to the standard output.
Example:

```
$ ctcat ~/etc.ct shadow
```

ctcp

This tool copies a file from a container to the file system or vice versa.
Example:

```
# Copy shadow from the container
$ ctcp ~/etc.ct shadow ~/shadow
# The same as above, with verbose output
$ ctcp -v ~/etc.ct shadow ~/shadow
# Copy /etc/init.d/rc into the container
$ ctcp -r /etc/init.d/rc rc
```

ctls

This tool lists all container files and, optionally, their metadata.
Example:

```
# Display the names of all files in the container
$ ctls ~/etc.ct
# Display the names and hashes of all files in the container
$ ctls -h ~/etc.ct
# Display the names and sizes of all files in the container
$ ctls -s ~/etc.ct
```

4. FUSE Container File System

There are legacy applications that use the POSIX API for file operations and can not be ported, because it would either be too much work or the source code is not available at all. To enable easy and transparent use of the containers via the POSIX API, a FUSE file system was created. Because of conflicts between the container library and the way programs usually write files, only read-only access was implemented.

4.1. Realization

The FUSE file system was realized as an overlay file system that makes the whole underlying file system accessible. Containers, however, are handled as directories and files within containers can be accessed like normal files in a directory. For example, if the FUSE file system was mounted at `/ctfs` and a container was available at `/storage/files.ct`, then all files within this container would be available in the directory `/ctfs/storage/files.ct`.

4.2. Implementation

FUSE provides an API to easily implement new FUSE file systems. In this section it is shown how a simple FUSE file system can be implemented in C.
The main work is done by the `fuse_main` function that handles command line parameters and the actual mounting of the file system. The user only has to implement the individual file system operations like `open`, `read`, `write` and `close`.

Listing 4.1: `main` function

```c
#include <fuse.h>

int main (int argc, char* argv[])
{
	return fuse_main(argc, argv, &ctfs_oper, NULL);
}
```

As can be seen in listing 4.1, a FUSE file system looks like any other C program. The `ctfs_oper` structure contains a mapping between file system operations and the functions implementing them.

Listing 4.2: `ctfs_oper` structure

```c
struct fuse_operations ctfs_oper = {
	.getattr	= ctfs_getattr,
	.init		= ctfs_init,
	.open		= ctfs_open,
};
```

Listing 4.2 shows the `ctfs_oper` structure containing three file system operations. The `init` operation is not a file system operation in the usual sense as it is called whenever the FUSE file system is mounted.

Listing 4.3: `ctfs_init` function

```
void* ctfs_init (struct fuse_conn_info* conn)
{
        struct ctfs_private_data* pd;

        pd = g_new(struct ctfs_private_data, 1);

        pd->containers = g_hash_table_new_full(g_str_hash,
            ↪ g_str_equal, ctfs_destroy_key, ctfs_destroy_value);

        return pd;
}
```

Listing 4.3 shows the `ctfs_init` function. The user may return a pointer to a memory address that will be made available to all other file system operations. This is used to keep a hash table of open containers in memory to speed up access.

Listing 4.4: `ctfs_open` function

```
int ctfs_open (const char* path, struct fuse_file_info* fi)
{
        char* dirname;
        char* basename;
        struct ct_container* cd;
        int ret;

        ret = -ENOENT;
        dirname = g_path_get_dirname(path);
        basename = g_path_get_basename(path);

        if ((cd = ctfs_get_container(dirname)) != NULL)
        {
                if (ct_file_exists(cd, basename))
                {
                        ret = 0;
                }
        }

        g_free(basename);
        g_free(dirname);

        if (ret == 0 && fi->flags & (O_RDWR | O_WRONLY))
        {
                ret = -EACCES;
        }

        return ret;
}
```

Listing 4.4 shows the `ctfs_open` function. The function checks whether the requested file exists and returns an appropriate return value. As can be seen, the last component of the path (`basename`) is treated as a file name while the rest (`dirname`) specifies the container. The `ctfs_get_container` function transparently manages the container hash table and returns a container handle. The existence of the file is checked with `ct_file_exists`. Additionally, the function returns an error if the file is not opened read-only.

Listing 4.5: `ctfs_get_container` function

```
struct ct_container* ctfs_get_container (const char* path)
{
        GHashTable* containers;
        struct ct_container* container;
        struct fuse_context* context;

        context = fuse_get_context();
        containers = ((struct
           ↪ ctfs_private_data*)context->private_data)->containers;

        if ((container = g_hash_table_lookup(containers, path)) ==
           ↪ NULL)
        {
                if ((container = ct_container_open(path)) != NULL)
                {
                        g_hash_table_insert(containers,
                           ↪ g_strdup(path), container);
                }
        }

        return container;
}
```

Listing 4.5 shows the `ctfs_get_container` function. The function gets passed a path to a container. If this container has not been opened yet, it is opened and its handle inserted into the hash table. Otherwise, the handle is taken directly from the hash table. The hash table is available as the `private_data` member (as specified by `ctfs_init`) of the so-called FUSE context that gets returned by `fuse_get_context()`.

Listing 4.6: `ctfs_getattr` function

```
int ctfs_getattr (const char* path, struct stat* stbuf)
{
        char* dirname;
        char* basename;
        struct ct_container* cd;
        struct ct_file* cf;
        int ret;

        /* FIXME: use lstat() here? */
        if (stat(path, stbuf) == 0)
        {
                ret = 0;

                if (stbuf->st_mode & S_IFREG)
                {
                        /* Fake a directory. */
                        stbuf->st_mode = (stbuf->st_mode & ~S_IFREG) | S_IFDIR | 0111;
                }
        }
        else
        {
                ret = -ENOENT;
                dirname = g_path_get_dirname(path);
                basename = g_path_get_basename(path);

                if ((cd = ctfs_get_container(dirname)) != NULL)
                {
                        if ((cf = ct_file_open(cd, basename)) != NULL)
                        {
                                ret = 0;
                                stat(dirname, stbuf);
                                stbuf->st_size = ct_file_size(cf);
                                ct_file_close(cf);
                        }
                }

                g_free(basename);
                g_free(dirname);
        }

        if (ret == 0)
        {
                /* We don't support writing at all. */
                stbuf->st_mode &= ~0222;
        }

        return ret;
}
```

Listing 4.6 shows the `ctfs_getattr` function. It is one of the most important functions in a

FUSE file system as it gets called before each access to a file. As a `stat()` replacement, it is supposed to fill a `stat` structure with the appropriate information if the file exists. Therefore, the `ctfs_getattr` function first calls `stat()` on the file to see if it exists. If the file is actually a directory, the information is passed on unmodified. However, if it is a file, we assume that it is a container and represent it as a directory by modifying the `st_mode` member of the `stat` structure. If the file does not exist on the underlying file system at all, we assume that the user wanted to open a file within a container. Thus, the last path component (`basename`) is used as the file name and the rest of the path (`dirname`) is assumed to be a container. If the file `basename` exists within the container `dirname` the `stat` structure is filled with the information of the container and only the `st_size` member is modified to reflect the actual file size. This probably is the biggest performance issue of this function, because `stat()` gets called twice. However, this can easily be corrected by storing an appropriate `stat` structure within the already existing hash table that is used as a cache.

5. Evaluation

5.1. Hardware

All benchmarks were run on a machine with one Intel Pentium M 1.6 GHz, 512 MB RAM and a 60 GB HDD.

5.2. Without Metadata

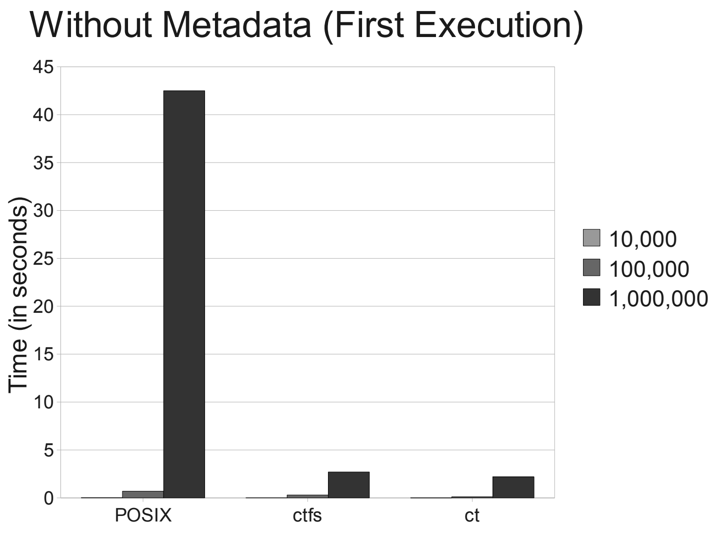

Figure 5.1.: Without metadata, first execution

Figure 5.1 shows numbers from benchmarks run on a freshly mounted file system, that is, with a cold cache. As can be seen, ctfs has almost no overhead when compared to ct, which is quite satisfying, considering the additional layer all file system calls have to go through.

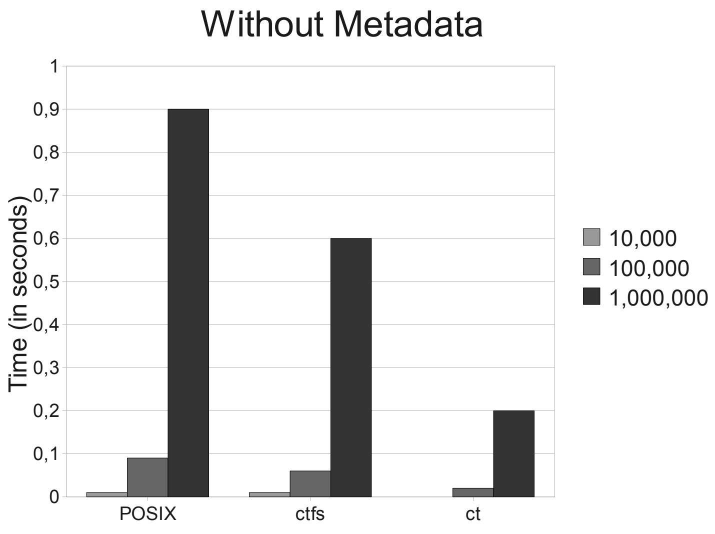

Figure 5.2.: Without metadata, subsequent executions

Figure 5.2 shows numbers from benchmarks run on the same file system as before without a preceding remount, that is, with a hot cache. POSIX, ctfs and ct are all faster here, because no data has to be read from the disk. It is interesting to note that the overhead of ctfs is now more visible. This is due to the fact that the overhead was negligible when actual disk access happened as in the previous benchmark.

5.3. With Metadata

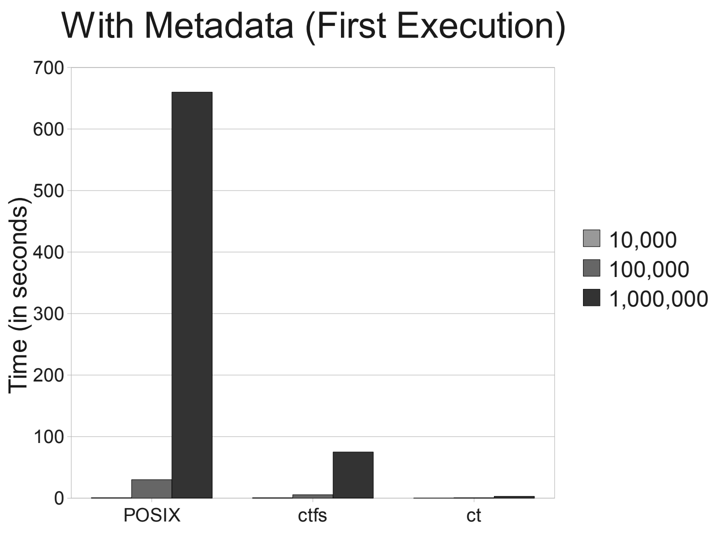

Figure 5.3.: With metadata, first execution

Figure 5.3 shows numbers from benchmarks run on a freshly mounted file system, that is, with a cold cache. As can be seen, ctfs is now much slower than ct here. This is probaby due to the fact that `ctfs_getattr` calls `stat()` two times. However, it should be possible to optimize this function quite easily. The speed of ct is around same as in the case without metadata, because all the metadata gets read from disk in any case.

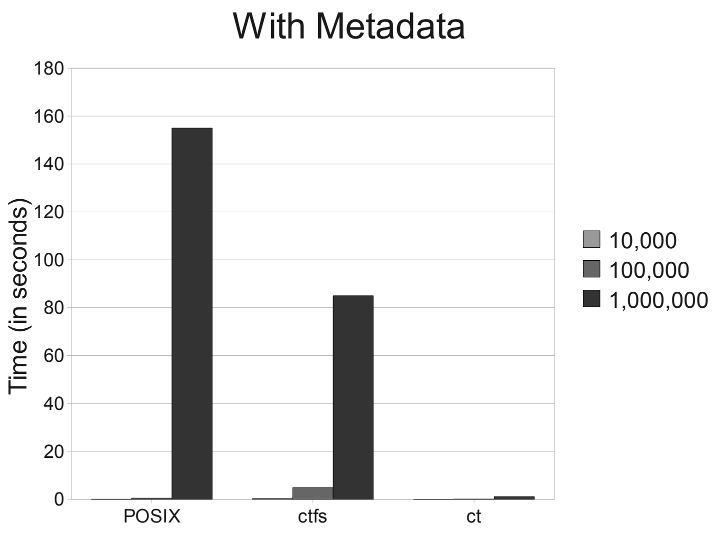

Figure 5.4.: With metadata, subsequent executions

Figure 5.4 shows numbers from benchmarks run on the same file system as before without a preceding remount, that is, with a hot cache. It is interesting to note that for 1,000,000 files ctfs is actually slower than in the first run with a cold cache. However, the reason for this is not apparent. Also, the speedup for POSIX is relatively small for 1,000,000 files. This is probably due to the small amount of RAM in the machine, which could therefore not cache all metadata which in turn had to be read from the disk intermittently.

A. Usage Instructions

A.1. Installing Required Packages

```
$ cd ct
$ sudo aptitude install libfuse-dev libglib2.0-dev libssl-dev
```

A.2. Compiling Everything

```
$ cd ct
$ make
```

A.3. Generating `libct` Documentation

```
$ cd ct
$ doxygen
```

A.4. FUSE file system

```
$ cd ct/ctfs
$ export LD_LIBRARY_PATH=../lib
$ ./ctfs ${MOUNTPOINT}
```

B. Benchmark Program

```
#!/bin/sh

set -x

remount ()
{
        sudo umount "${1}"
        sudo mount "${1}"
}

unmount_ctfs ()
{
        sleep 1
        fusermount -u "${1}"
}

mount_ctfs ()
{
        "${CT}/ctfs/ctfs" "${1}"
}

[ -z "${1}" ] && exit 1
[ -z "${2}" ] && exit 1

SIZE="${1}"
COUNT="${2}"

MOUNT="/foo"
ROOT="/foo/bar"
FUSE_MOUNT="${HOME}/ctfs"
CT="${HOME}/ct"
LOG="${HOME}/${SIZE}/${COUNT}"

export LD_LIBRARY_PATH="${CT}/lib"

[ ! -d "${ROOT}/${SIZE}/${COUNT}" ] && exit 1

mkdir -p "${LOG}"
unmount_ctfs "${FUSE_MOUNT}"

```

```
# POSIX
for mode in meta no_meta
do
        [ "${mode}" = "meta" ] && META="-m" || META=""

        for run in 0 1 2
        do
                remount "${MOUNT}"

                for i in 0 1 2
                do
                        (time "${CT}/tools/benchmark" -p ${META} "${ROOT}/${SIZE}/${COUNT}") > /dev/null 2>> "${LOG}/posix-${mode}-${run}.txt"
                done
        done
done

# ct
for mode in meta no_meta
do
        [ "${mode}" = "meta" ] && META="-m" || META=""

        for run in 0 1 2
        do
                remount "${MOUNT}"

                for i in 0 1 2
                do
                        (time "${CT}/tools/benchmark" -c ${META} "${ROOT}/${SIZE}/${COUNT}.ct") > /dev/null 2>> "${LOG}/ct-${mode}-${run}.txt"
                done
        done
done

# ctfs
for mode in meta no_meta
do
        [ "${mode}" = "meta" ] && META="-m" || META=""

        for run in 0 1 2
        do
                unmount_ctfs "${FUSE_MOUNT}"
                remount "${MOUNT}"
        mount_ctfs "${FUSE_MOUNT}"

        for i in 0 1 2
        do
            (time "${CT}/tools/benchmark" -p ${META} "${FUSE_MOUNT}/${ROOT}/${SIZE}/${COUNT}.ct") > /dev/null 2>> "${LOG}/ctfs-${mode}-${run}.txt"
        done
    done
done

unmount_ctfs "${FUSE_MOUNT}"
```

C. BSD License

D. Benchmark Data

Without Metadata

First Execution

	10,000	100,000	1,000,000
POSIX	0.1s	0.7s	41s
ctfs	0.05s	0.3s	2.7s
ct	0.00s	0.1s	2.2s

Subsequent Executions

	10,000	100,000	1,000,000
POSIX	0.01s	0.1s	0.9s
ctfs	0.01s	0.06s	0.6s
ct	0.00s	0.03s	0.2s

With Metadata

First Execution

	10,000	100,000	1,000,000
POSIX	0.6s	29.5s	12m15s
ctfs	0.5s	5.5s	1m16s
ct	0.01s	0.5s	3.1s

Subsequent Executions

	10,000	100,000	1,000,000
POSIX	0.05s	0.5s	2m35s
ctfs	0.3s	5.2s	1m28s
ct	0.01s	0.1s	1.1s

Bibliography

[Hei07] Hendrik Heinrich. Ein Container-Format für den wahlfreien effizienten Zugriff auf Dateien. Bachelor's Thesis, Ruprecht-Karls-Universität Heidelberg, September 2007.