memfs – A FUSE Memory File System

Softwarepraktikum für Fortgeschrittene

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

Michael Kuhn
Betreuer: Julian Kunkel

2008-10-12

Contents

1. Introduction

1.1. Goals

The goal of this practical was to measure the overhead of the FUSE[1] framework, because previous research indicates that FUSE introduces significant overhead when a large number of files is processed. FUSE file systems themselves run in user space and use the special device `/dev/fuse` to communicate with the kernel part of FUSE. Because of this more expensive context switches have to be performed.

1.2. Requirements

To eliminate the influence of the relatively slow hard disk, the file system was implemented as a memory file system, much like `tmpfs`[2].

[1] Filesystem in Userspace – `http://fuse.sourceforge.net/`
[2] `http://en.wikipedia.org/wiki/TMPFS`

2. FUSE Memory File System

2.1. 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. To make the implementation easier, GLib[1] was used. All data types beginning with `G` and all functions beginning with `g_` belong to GLib.

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 2.1: `main` function

```
1  #define FUSE_USE_VERSION 26
2  #include <fuse.h>
3
4  int main (int argc, char* argv[])
5  {
6          return fuse_main(argc, argv, &memfs_oper, NULL);
7  }
```

As can be seen in listing 2.1, a FUSE file system looks like any other C program. The header file `fuse.h` contains all necessary declarations. `FUSE_USE_VERSION` can be used to specify the actual API version to use. The `memfs_oper` structure contains a mapping between file system operations and the functions implementing them.

Listing 2.2: `memfs_oper` structure

```
1  struct fuse_operations memfs_oper = {
2          .chmod    = memfs_chmod,
3          .chown    = memfs_chown,
4          .create   = memfs_create,
5          .destroy  = memfs_destroy,
6          .getattr  = memfs_getattr,
7          .init     = memfs_init,
8          .link     = memfs_link,
9          .mkdir    = memfs_mkdir,
10         .open     = memfs_open,
11         .read     = memfs_read,
12         .readdir  = memfs_readdir,
13         .rmdir    = memfs_rmdir,
14         .statfs   = memfs_statfs,
15         .truncate = memfs_truncate,
16         .unlink   = memfs_unlink,
17         .utimens  = memfs_utimens,
18         .write    = memfs_write,
19 };
```

[1] `http://www.gtk.org/`

Listing 2.2 shows the `memfs_oper` structure containing all implemented file system operations. The `init` and `destroy` operations are not a file system operation in the usual sense as they are called whenever the FUSE file system is mounted and unmounted. The following file system operations are merely empty stubs to make `fileop` run: `chmod`, `chown`, `open` and `utimens`.

Listing 2.3: `memfs_init` function

```
1 void* memfs_init (struct fuse_conn_info* conn)
2 {
3         struct memfs* fs;
4
5         fs = memfs();
6         fs->root = memfs_entry_new(entry_directory);
7
8         return fs;
9 }
```

Listing 2.3 shows the `memfs_init` function. The user may return a pointer to a memory address that will be made available to all other file system operations via the `private_data` member of the structure returned by `fuse_get_context()`. The `memfs_entry` structure is a wrapper around the `memfs_directory` and `memfs_file` structures to allow both types as entries within an directory.

Listing 2.4: `memfs` structures

```
1  struct memfs_directory
2  {
3  #if defined(MEMFS_FLAVOR_HASH_TABLE)
4          GHashTable* entries;
5  #elif defined(MEMFS_FLAVOR_TREE)
6          GTree* entries;
7  #endif
8  };
9
10 struct memfs_file
11 {
12         guint ref_count;
13         gchar* data;
14         goffset size;
15 };
16
17 struct memfs_entry
18 {
19         gint type;
20
21         union
22         {
23                 struct memfs_directory* fs_dir;
24                 struct memfs_file* fs_file;
25         }
26         e;
27 };
```

Listing 2.4 shows the `memfs` structures. As can be seen, the `memfs_directory` structure supports different data types to manage its entries. Currently, hash tables and balanced binary trees are supported. The `memfs_file` structure contains a reference counter to support multiple links to the same file, that is, a file can be made available under different paths. See

`memfs_link()` for more information. The `type` member of the `memfs_entry` structure indicates whether the entry is a directory or a file and therefore which pointer of the `e` member must be used.

Listing 2.5: `memfs_create` function

```
1  int memfs_create (const char* path, mode_t mode, struct fuse_file_info* fi)
2  {
3          int ret = -ENOENT;
4          struct memfs_entry* fs_entry;
5          char* dirname;
6
7          dirname = g_path_get_dirname(path);
8
9          if ((fs_entry = memfs_path_get_last_component(dirname)) != NULL)
10         {
11                 if (fs_entry->type == entry_directory)
12                 {
13                         memfs_directory_entry_insert(fs_entry->e.fs_dir,
                               ↪ g_path_get_basename(path),
                               ↪ memfs_entry_new(entry_file));
14                         ret = 0;
15                 }
16         }
17
18         g_free(dirname);
19
20         return ret;
21 }
```

Listing 2.5 shows the `memfs_create` function. The function creates the file given by the `path` argument. As can be seen, the last component of the path is treated as a file name while the rest (`dirname`) specifies the directory in which the file is created. The `memfs_path_get_last_component()` function checks each component of the path and returns a pointer to the `memfs_entry` structure that represents the last one. It is then checked if the last component is in fact a directory. If this is the case, a new file is created.

Listing 2.6: `memfs_path_get_last_component` function

```
1  struct memfs_entry* memfs_path_get_last_component (const gchar* path)
2  {
3          struct memfs* fs = memfs();
4          char** components;
5          gint i;
6          guint length;
7          struct memfs_entry* fs_entry;
8
9          components = g_strsplit(g_path_skip_root(path), G_DIR_SEPARATOR_S, 0);
10         length = g_strv_length(components);
11
12         fs_entry = fs->root;
13
14         for (i = 0; i < length; i++)
15         {
16                 struct memfs_entry* lookup_entry;
17
18                 if (fs_entry->type != entry_directory)
```

```
19                  {
20                          goto error;
21                  }
22
23                  if ((lookup_entry =
                        ↪ memfs_directory_entry_lookup(fs_entry->e.fs_dir,
                        ↪ components[i])) == NULL)
24                  {
25                          goto error;
26                  }
27
28                  fs_entry = lookup_entry;
29          }
30
31          g_strfreev(components);
32
33          return fs_entry;
34
35  error:
36          g_strfreev(components);
37
38          return NULL;
39  }
```

Listing 2.6 shows the `memfs_path_get_last_component` function. The function gets passed a path to check. It then checks every component of the path in two steps:

1. It is checked if the component is a directory
2. The next component is looked up in the list of this component's entries

As this function has to check each component and has to do a lookup for every single one of them, it is the slowest function of `memfs`. Because it has to be called for almost all file operations, it also heavily influences the overall performance. It has a complexity of $O(n)$ where n is the number of components of the path, that is, the number of subdirectories.

Listing 2.7: `memfs_getattr` function

```
1   int memfs_getattr (const char* path, struct stat* stbuf)
2   {
3           int ret = -ENOENT;
4           struct memfs_entry* fs_entry;
5
6           ...
7
8           if ((fs_entry = memfs_path_get_last_component(path)) != NULL)
9           {
10                  stbuf->st_mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
11                  stbuf->st_nlink = 1;
12                  stbuf->st_uid = getuid();
13                  stbuf->st_gid = getgid();
14                  stbuf->st_atime = stbuf->st_mtime = stbuf->st_ctime = 0;
15
16                  switch (fs_entry->type)
17                  {
18                          case entry_directory:
```

```
19                                      stbuf->st_mode |= S_IFDIR | S_IXUSR | S_IXGRP |
                                            ↪ S_IXOTH;
20                                      stbuf->st_size =
                                            ↪ memfs_directory_size(fs_entry->e.fs_dir);
21
22                                      ret = 0;
23                                      break;
24                              case entry_file:
25                                      stbuf->st_mode |= S_IFREG;
26                                      stbuf->st_size = fs_entry->e.fs_file->size;
27
28                                      ret = 0;
29                                      break;
30                      }
31              }
32
33              return ret;
34      }
```

Listing 2.7 shows the `memfs_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.

2.1.1. `/opts` Configuration Directory

The `/opts` directory allows the `memfs` file system to be configured at runtime much like the `/proc` file system in Unix-like operating systems. This is implemented as a fake directory in the root directory of the `memfs` file system. The `/opts` directory in turn contains files that can be read and written to view and change the configuration.

Listing 2.8: `memfs_readdir` function

```
1   int memfs_readdir (const char* path, void* buf, fuse_fill_dir_t filler, off_t
        ↪ offset, struct fuse_file_info* fi)
2   {
3           int ret = -ENOENT;
4           struct memfs_entry* fs_entry;
5
6           if (G_UNLIKELY(path[1] == '\0'))
7           {
8                   filler(buf, "opts", NULL, 0);
9           }
10          else if (G_UNLIKELY(strcmp(path + 1, "opts") == 0))
11          {
12                  filler(buf, "no_data", NULL, 0);
13
14                  return 0;
15          }
16
17          ...
18  }
```

Listing 2.8 shows the relevant part of the `memfs_readdir` function. If a listing of the root directory is requested, it injects the `opts` directory into the listing. If a listing of the `/opts` directory is requested, it returns a listing of the configuration files. Currently only the `no_data` setting is exposed in the `/opts` directory. The `G_UNLIKELY` macro helps the compiler optimize

the execution of the `if` statement. This is done, because these statements should not impact performance too much.

Listing 2.9: `memfs_read` function

```
1  int memfs_read (const char* path, char* buf, size_t size, off_t offset, struct
       ↪ fuse_file_info* fi)
2  {
3          struct memfs* fs = memfs();
4          int ret = -ENOENT;
5          struct memfs_entry* fs_entry;
6
7          if (G_UNLIKELY(strncmp(path, "/opts/", 6) == 0))
8          {
9                  if (strcmp(path + 6, "no_data") == 0)
10                 {
11                         ret = 0;
12
13                         switch (offset)
14                         {
15                                 case 0:
16                                         memcpy(buf, (fs->opts.no_data) ? "1" :
                                               ↪ "0", 1);
17                                         ret++;
18                                 case 1:
19                                         memcpy(buf + ret, "\n", 1);
20                                         ret++;
21                         }
22                 }
23
24                 return ret;
25         }
26
27         ...
28 }
```

Listing 2.9 shows the relevant part of the `memfs_read` function. If the `/opts/no_data` file is read, it simply returns the boolean value of the `no_data` option as a string.

Listing 2.10: `memfs_write` function

```
1  int memfs_write (const char* path, const char* buf, size_t size, off_t offset,
       ↪ struct fuse_file_info* fi)
2  {
3          struct memfs* fs = memfs();
4          int ret = -ENOENT;
5          struct memfs_entry* fs_entry;
6
7          if (G_UNLIKELY(strncmp(path, "/opts/", 6) == 0))
8          {
9                  if (strcmp(path + 6, "no_data") == 0)
10                 {
11                         fs->opts.no_data = (buf[0] == '1');
12                         ret = size;
13                 }
14
15                 return ret;
16         }
```

```
17
18          ...
19  }
```

Listing 2.10 shows the relevant part of the `memfs_write` function.

2.2. Complex Operations

Some of FUSE's file system operations are complex, that is, are internally made up of several file system operations.

2.2.1. `setattr()`

After each of the operations `chmod()`, `chown()`, `truncate()` and `utimens()` an implicit `getattr()` is performed on the selected path. Several of these operations are grouped together as one `setattr()` if they are performed on the same path. The `getattr()` is performed in the same atomic operation as the `setattr()`.[2]

2.2.2. `lookup()`

After each of the operations `create()`, `mknod()`, `mkdir()`, `symlink()`, and `link()` an implicit `getattr()` is performed on the affected path. The `getattr()` is performed in the same atomic operation as the original operation.

[2]This behavior can be disabled. Operations can then be interrupted.

3. Evaluation

The `fileop` tool from the IOzone Filesystem Benchmark[1] was used to conduct several benchmarks. `memfs` and `tmpfs` were compared, each with a varying number of files. The raw data of the figures presented here can be found in appendix B.

3.1. Hardware

All benchmarks were run on a machine with one Intel Pentium M 1.6 GHz and 512 MB RAM running Linux 2.6.27 and FUSE 2.7.3.

3.2. File Number

To analyze how the different file systems compare against each other, the following figures show for 125,000, 512,000 and 1,000,000 files the number of operations per second with each file system. It is important to note that only 2,550, 6,480 and 10,100 directories are created for the file numbers above, therefore the results for `mkdir`, `rmdir` and `readdir` may not be as accurate.

[1] `http://www.iozone.org/`

3.2.1. 125,000 Files

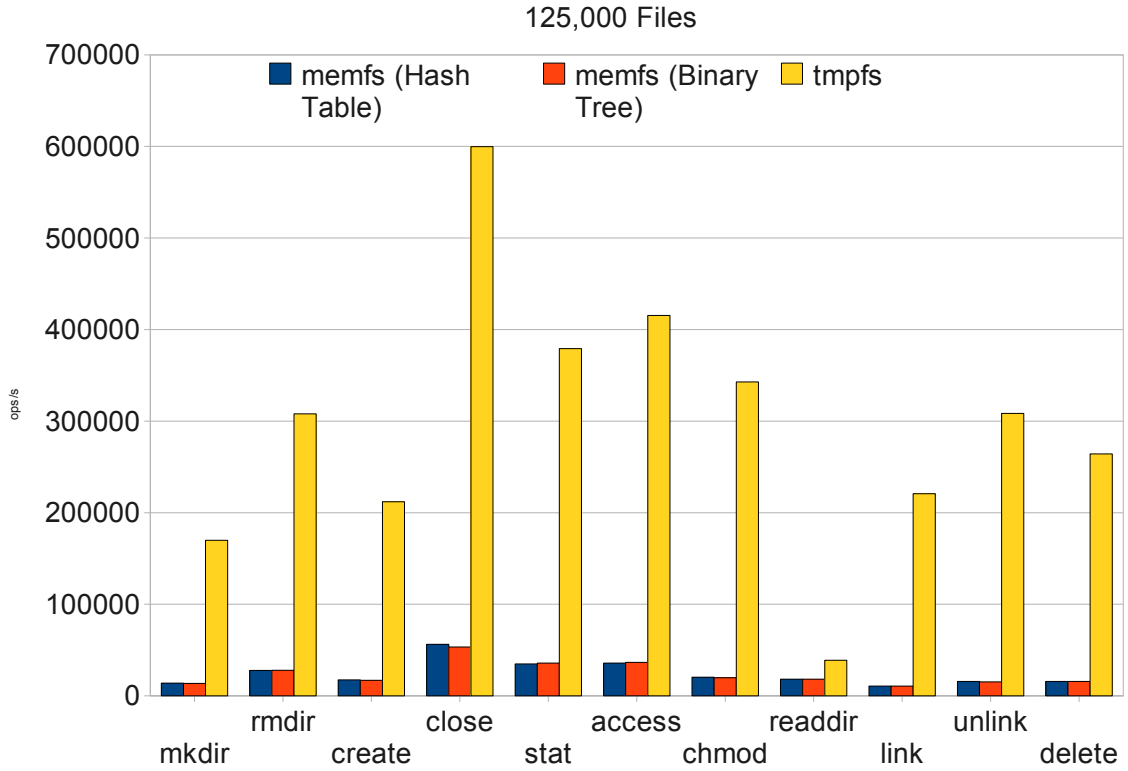

Figure 3.1.: 125,000 Files

Figure 3.1 shows a comparison of `memfs` and `tmpfs` for 125,000 files. `memfs` is used with hash table and binary tree backends. As can be seen, the results for `memfs` are about the same for the hash table and binary tree backends. `tmpfs` however is much faster, because it has to do less expensive context switches. Overall, all `tmpfs` operations are 10–20 times faster than those in `memfs`. The `close()` (that is, `release()`) operation reaches the most operations per second with about 56,000 operations per second. Since the operation is not implemented and does no additional implicit work it can be considered as an approximate maximum.

3.2.2. 512,000 Files

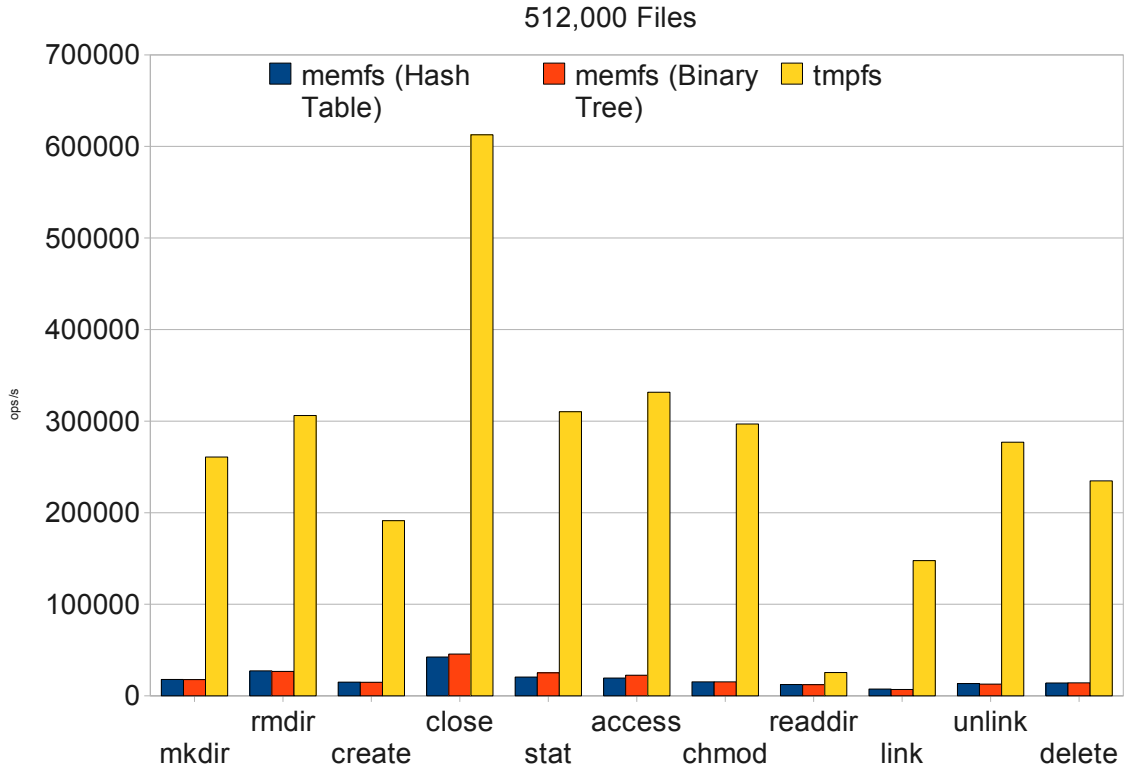

Figure 3.2.: 512,000 Files

Figure 3.2 shows a comparison of `memfs` and `tmpfs` for 512,000 files. `memfs` and `tmpfs` show the same behavior as in figure 3.1. Overall, all `tmpfs` operations are more than 10–20 times faster than those in `memfs`. The `close()` (that is, `release()`) operation reaches the most operations per second with about 42,000 operations per second. It is important to note that this is slower than with 125,000 files. This is probably due to the fact that FUSE filled the kernel cache in this case and therefore the kernel had to do more clean-up work, decreasing FUSE's performance.

3.2.3. 1,000,000 Files

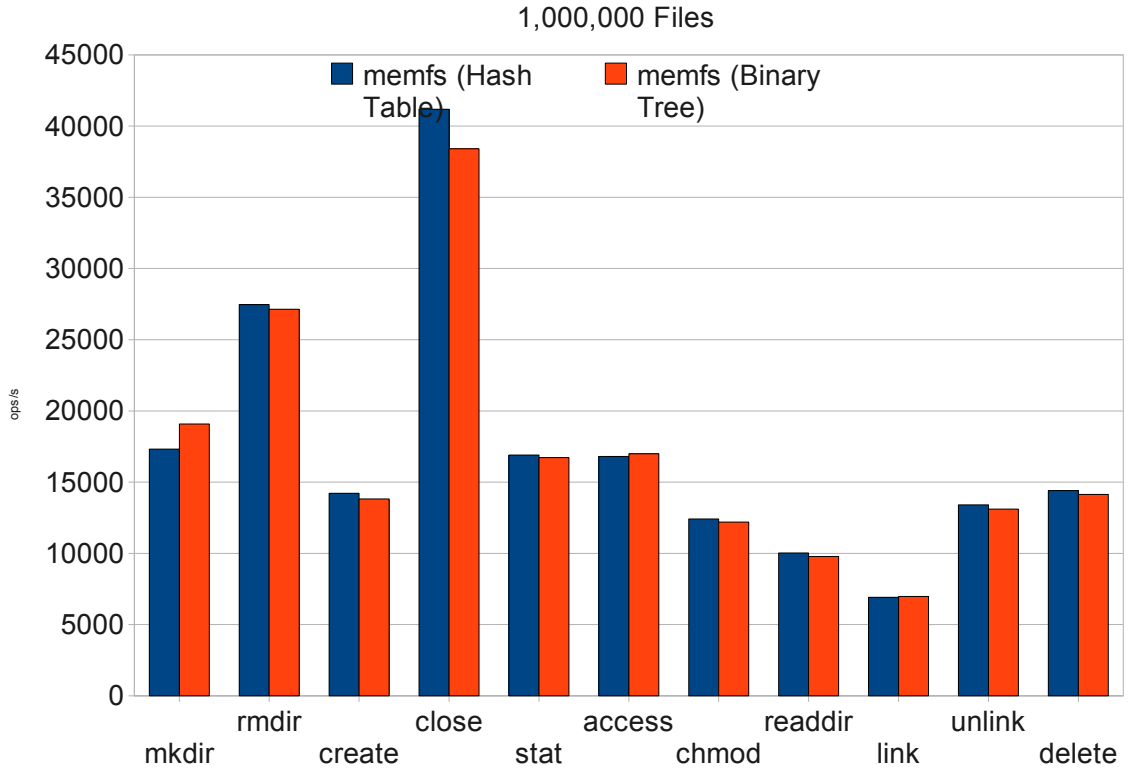

Figure 3.3.: 1,000,000 Files

Figure 3.3 shows a comparison for 1,000,000 files. `memfs` shows the same behavior as in figure 3.1. There is no result for `tmpfs`, because it could not store 1,000,000 files in 512 MB of RAM.[2] The `close()` (that is, `release()`) operation reaches the most operations per second with about 41,000 operations per second. It is important to note that this is slower than with 125,000 files, but not much slower than with 512,000 files.

3.3. File System

To analyze how the different file systems handle increasing amounts of files, the following figures show for each file system the number of operations per second with 125,000, 512,000 and 1,000,000 files.

[2] Apparently, it ran out of memory, even with `tmpfs`'s memory limit. The kernel started killing processes, effectively crashing the whole machine.

3.3.1. `memfs` (Hash Table)

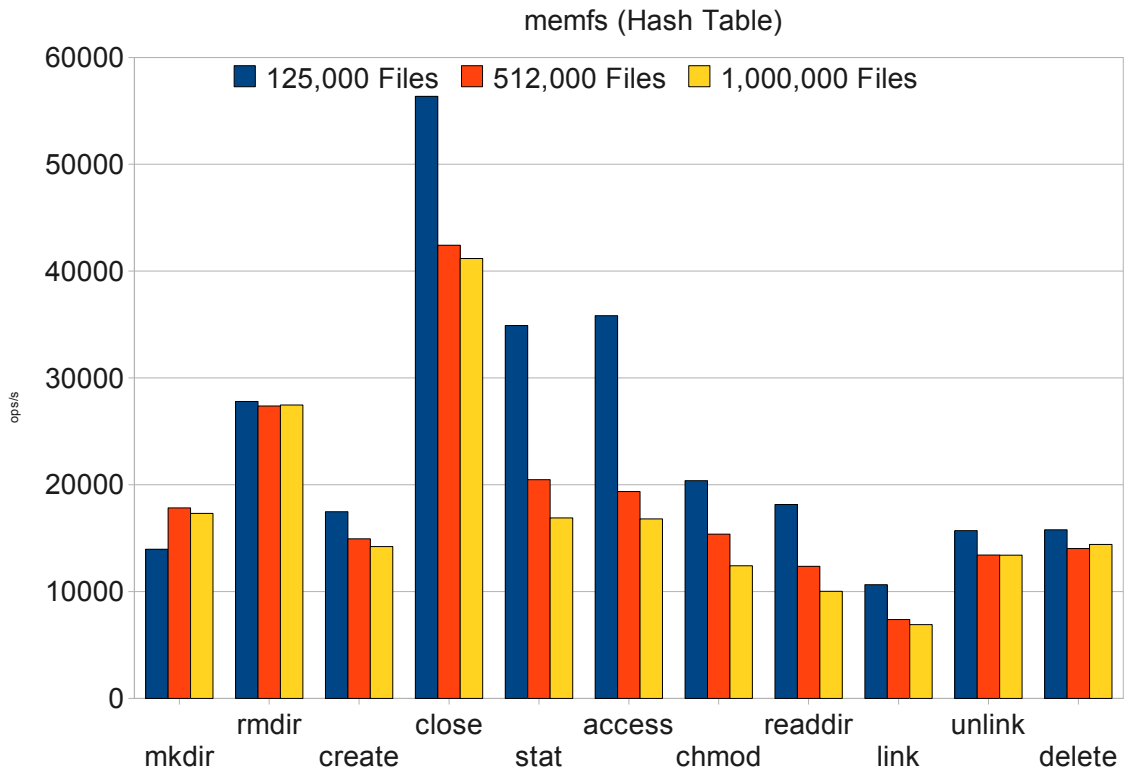

Figure 3.4.: memfs (Hash Table)

Figure 3.4 shows the results of `memfs` when configured to use a hash table as backend. As can be seen there is a large drop in performance when increasing the number of files from 125,000 to 512,000. An increase from 512,000 to 1,000,000 files causes a smaller performance decrease. It is not clear why this is the case. In some cases performance stays the same.

3.3.2. `memfs` (Binary Tree)

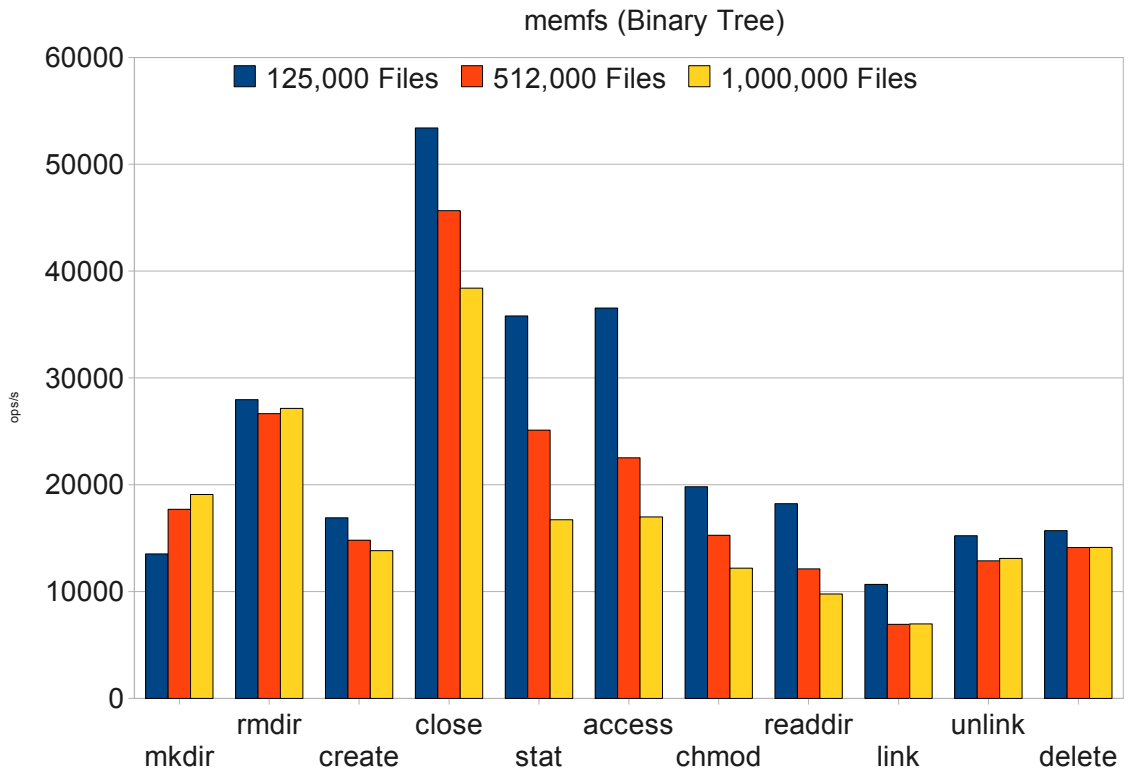

Figure 3.5.: memfs (Binary Tree)

Figure 3.5 shows the results of `memfs` when configured to use a balanced binary tree as backend. As can be seen there is a drop in performance when increasing the number of files from 125,000 to 512,000. However, this drop is not as big as in figure 3.4. An increase from 512,000 to 1,000,000 files causes an equally big performance decrease. Again, in some cases performance stays the same.

3.3.3. `tmpfs`

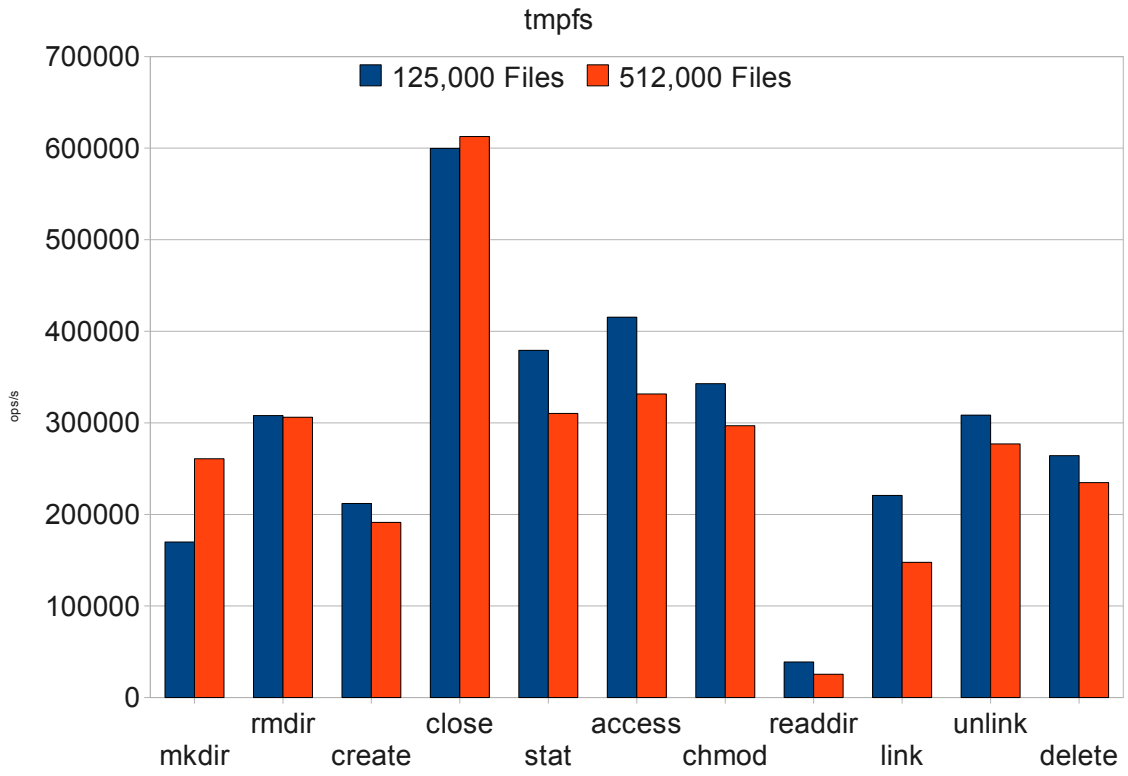

Figure 3.6.: tmpfs

Figure 3.6 shows the results of `tmpfs`. As can be seen there is a drop in performance when increasing the number of files from 125,000 to 512,000.

4. Conclusion and Future Work

4.1. `memfs`

With `memfs` there now exists a memory file system that is configurable at runtime and can be easily extended to use arbitrary data structures as backends. This will allow benchmarking and – hopefully – tuning of FUSE with large amounts of files. However, some problems have to be considered first.

4.2. FUSE Overhead

Currently, it is hard to measure the overhead with empty stub operations, because FUSE performs implicit `getattr()` calls for most of them. `release()` is one of the few operations that can be used. Since it is not implemented in `memfs` it introduces no overhead of its own. Therefore, its performance should give a good estimate of the possible maximum that FUSE is capable of. It would be interesting to modify the FUSE user-space library to make the implicit `getattr()` calls conditional. Since this is apparently only done to populate the caches, it should be safe to disable this behavior.

A. Usage Instructions

A.1. Installing Required Packages

```
$ sudo aptitude install libfuse-dev libglib2.0-dev
```

A.2. Compiling Everything

```
$ cd memfs
$ make
```

A.3. FUSE File System

```
$ cd memfs
$ ./memfs ${MOUNTPOINT}
```

A.3.1. Debugging

To debug the FUSE file system, use the `-f` argument.

```
$ cd memfs
$ gdb --args ./memfs -f ${MOUNTPOINT}
```

The `-d` argument causes FUSE to print debug output. Do **not** run benchmarks with this.

```
$ cd memfs
$ ./memfs -d ${MOUNTPOINT}
```

A.4. Benchmark

```
$ cd memfs
$ fileop -f {50,80,100} -s 0
```

B. Benchmark Data

B.1. 125,000 Files

The following table shows operations per second.

	memfs (Hash Table)	memfs (Binary Tree)	tmpfs
mkdir	13,964	13,553	169,864
rmdir	27,805	27,964	307,899
create	17,484	16,912	211,971
read	507,788	465,898	556,355
write	541,531	532,712	696,359
close	56,360	53,401	599,711
stat	34,910	35,807	379,100
access	35,833	36,547	415,415
chmod	20,380	19,822	342,789
readdir	18,153	18,234	38,874
link	10,653	10,680	220,692
unlink	15,711	15,241	308,403
delete	15,795	15,716	264,165

B.1.1. Memory Usage

As reported by `top`.

memfs (Hash Table)	memfs (Binary Tree)	tmpfs
43 MB	47 MB	n/a

B.2. 512,000 Files

The following table shows operations per second.

	memfs (Hash Table)	memfs (Binary Tree)	tmpfs
mkdir	17,832	17,711	260,776
rmdir	27,372	26,660	306,112
create	14,944	14,804	191,267
read	502,365	506,404	697,742
write	583,254	554,159	670,165
close	42,422	45,656	612,714
stat	20,476	25,121	310,329
access	19,366	22,527	331,634
chmod	15,374	15,276	296,911
readdir	12,367	12,126	25,415
link	7,386	6,941	147,648
unlink	13,429	12,882	276,924
delete	14,034	14,123	234,755

B.2.1. Memory Usage

As reported by `top`.

memfs (Hash Table)	memfs (Binary Tree)	tmpfs
119 MB	133 MB	n/a

B.3. 1,000,000 Files

The following table shows operations per second.

	memfs (Hash Table)	memfs (Binary Tree)	tmpfs
mkdir	17,322	19,084	n/a
rmdir	27,468	27,144	n/a
create	14,219	13,828	n/a
read	498,137	506,167	n/a
write	571,636	590,069	n/a
close	41,182	38,400	n/a
stat	16,904	16,723	n/a
access	16,804	16,997	n/a
chmod	12,419	12,194	n/a
readdir	10,032	9,778	n/a
link	6,915	6,981	n/a
unlink	13,403	13,106	n/a
delete	14,413	14,142	n/a

1 Byte Files

	memfs (Hash Table)
mkdir	18,589
rmdir	27,813
create	16,680
read	20,029
write	38,426
close	33,651
stat	19,504
access	17,792
chmod	12,775
readdir	10,088
link	6,954
unlink	14,235
delete	14,728

B.3.1. Memory Usage

As reported by `top`.

memfs (Hash Table)	memfs (Binary Tree)	tmpfs
191 MB	199 MB	n/a