A Stackable Caching File System: Simple XCacheFS

Anunay Gupta, Sushma Uppala, Yamini Pradeepthi Allu
Stony Brook University Computer Science Department
Stony Brook, NY 11794-4400
{anunay, suppala, yaminia}@cs.sunysb.edu

ABSTRACT

Stackable file systems are modular file systems that can be mounted to enhance the functionality of the underlying file systems. Here, we present the design and implementation of a stackable fan-out caching file system, which caches files and directories in a source branch to a cache branch in a flat hierarchy. Typically, the Source Branch will come from a slower file system (e.g. a Network File System) and Cache Branch will come from a faster file system (local disk based file system). Apart from making file access faster through caching files, the flat hierarchy of the cache branch simplifies book-keeping of the content, hence making it easier to administer the cached files.

KEYWORDS
Simple XCacheFS; Flat file systems; Stackable File Systems; Linux File Systems.

1. INTRODUCTION

The main challenge in implementing a flat hierarchical cache is to achieve a bidirectional mapping of filenames in the Source Branch (SB) to unique filenames in Cache Branch (CB). We present a design to handle this bidirectional mapping so that files in CB can be listed and accessed as if a directory hierarchy exists even when SB is not available. The rest of the report is organized as follows. Section 2 gives a brief description of the background of the stackable and fan-out file systems. Section 3 presents the design of Simple XCacheFS. Section 4 gives the implementation details of the relevant file system operations. In Section 5, the performance of the Simple XCacheFS is evaluated with respect to that of a stackable caching file system which replicates the hierarchy of files and directories in the SB to the CB.

2. BACKGROUND

2.1. Stackable file systems

To add additional functionality to any file system instead of modifying it, a new file system layer can be stacked on top of an existing one. This makes the functionality modular and can be ported to different file systems without much effort.

A stackable file system when mounted on an already mounted file system, intercepts the calls from VFS to lower file system operations to provide additional functionality. Hence it presents itself to VFS as a traditional file system and to the lower file system as VFS. This way various file systems can be stacked on top of each other, each adding a specific functionality. This implementation of Simple XCacheFS is based on FiST: A Language for Stackable File Systems [1].

2.2. Fan-out file systems

The stackable file system can be mounted over two or more lower file systems to produce interesting file systems: replication, fail-over, caching, load-balancing, unification, and more.

XCacheFS is one of the several useful implementations of stackable fan-out file systems. In a conventional caching file system [2], the directory hierarchy in a source branch is replicated as it is into the cache branch whenever a file is cached, unlike Simple XCacheFS. This would make the implementation simple, but caching of files in a flat hierarchy is useful in certain file systems to simplify implementation of eviction policies.

Figure 2.1 shows a high level structure of Simple XcacheFS file system.

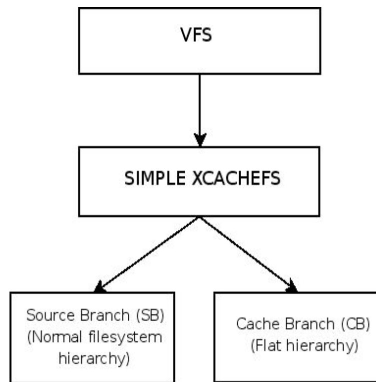

Figure 2.1: Stackable fan-out Simple-XcacheFS

3. DESIGN

3.1. Encoding and Decoding of Unique File Names

To achieve a flat hierarchy in Cache Directory every file in Source branch has to be mapped to a unique file name in Cache Branch.

In this project, a unique file name in CB is obtained by hashing the absolute path name of the file in SB. Since the absolute path of any file is unique, the hash of it would produce a unique value. We use MD5 algorithm to hash the absolute path name. Hence, any file in CB would default to a filename of length 32 bytes, which is less than the NAME_MAX of 256 bytes conforming to the semantics of UNIX file naming convention. Since collision rates in hashing are infinitesimally less, we can assume that hashing a unique pathname or an existing unique hash produces a unique hash.

This provides a unidirectional way to encode unique file names. Since hashes are irreversible, we cannot decode the encoded file names to obtain the absolute path name of the file/directory in SB.

The reverse mapping of unique filenames back to directory paths is possible through certain attributes of a file. The Inode number, Inode generation number, File System ID if encoded together will be unique for any file on a system. This unique file handle can be easily reversed to obtain the absolute pathname of the file in the Source Branch by using the file system objects of SB. This approach will fail if the corresponding file system objects of SB are not available, e.g., when SB is offline or not mounted.

To support access of Cache Branch when Source Branch is offline, we need to be able to decode the pathname of the file in SB using the file system objects of the CB. A possible solution is to maintain a bidirectional mapping of unique file handles in CB to absolute pathnames of files in SB. Moreover, this mapping must be stored persistently to maintain consistent state over remounts. One approach is to write this mapping to a file in CB. But, this would mean locking of this file to access information of any file in CB. As a result of this, two processes would need to synchronize to access any file in CB even if they are accessing different files. This could be a bottleneck and thus, impact the performance of the file system.

A better approach could be to split this mapping into different files. This is achieved implicitly in this project as every directory is cached as a file which contains the attributes of all the files and subdirectories contained in it and are cached. Hence this file is used to obtain a mapping back to the original filenames in CB.

3.2. Flat Hierarchical Representation of Cache

The Simple XcacheFS File System achieves a flat hierarchy in the representation of the cached files and directories. To achieve this, both the directories and the files are cached equally. Every directory in SB that is cached in CB is represented by a file (referred to as "*Pseudo Directory*" from now) which contains dirents of the files that are under it and are cached. The dirents contained in the pseudo directory have a special structure defined by Simple XcacheFS representing the actual *File Name*, *Inode Number* and *Type* of the file in the source branch. Figure 3.1 shows an example of hierarchy of files and directories in SB and the corresponding representation in CB.

For each operation on any of the files or directories that are cached (file/directory creation, rename or removal) the dirent record in the parent directory (which is also cached) is updated. Since all the directories and files are represented in a flat

hierarchy under CB, there should be a way to keep this record for the CB's root directory as well. Hence a separate file is created for CB during mount which will contain dirents representing files/directories that are directly under SB's root directory and are cached.

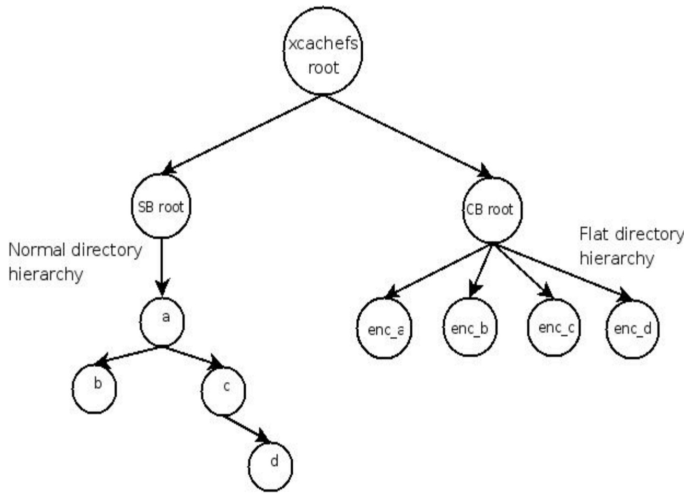

Figure 3.1: Directory hierarchy in Source and Cache

Though directories are created as files in CB, Simple XcacheFS hides this detail from the VFS, by maintaining the actual type of the file in the xcachefs level dentry, and assigning the xcachefs level inode operations accordingly. Further details on the implementation can be found in Section 4.

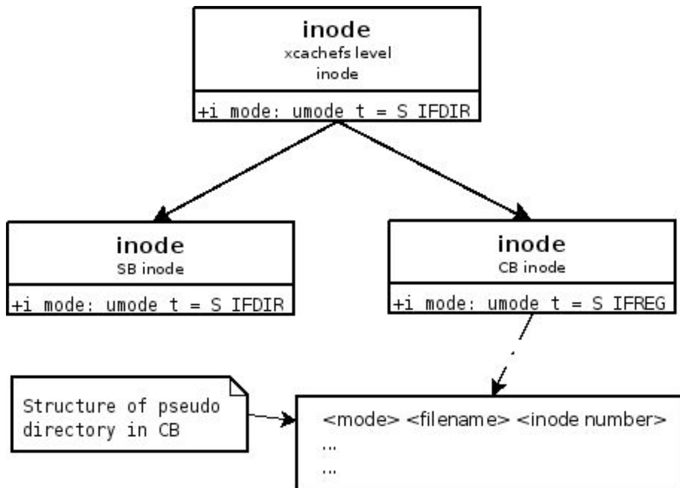

Figure 3.2: A directory in Source Branch is represented as a file "*pseudo directory*" in Cache Branch. XCacheFS keeps the type hidden from VFS.

4. IMPLEMENTATION

This section discusses the implementation details of certain important operations that need to be handled differently in case of a flat hierarchy cache.

4.1. Mounting Simple XCacheFS file system

Being a stackable fan-out file system, Simple XCacheFS can be mounted over a cache branch (CB) and/or the source branch (SB). We support two modes for mount.

i. *Both SB and CB are available.*

A typical mount command for this would be:

```
#mount-t xcachefs -o sb=/n/sb, cb=/n/cb
xcachefs /mnt/xcachefs
```

ii. *Only CB is available and SB is offline.*

This case is useful when the user loses connection to SB, but would want to access/modify/change the files in the cache while SB is offline. A typical mount command for this would be:

```
#mount -t xcachefs -o sb=NA, cb=/n/cb
xcachefs /mnt/xcachefs
```

Simple XCacheFS supports Offline mode only if the same CB was used for some SB and was mounted along with it earlier i.e. the cache has some files cached and enough information was logged to map the encoded names back to file names in SB.

Method: `xcachefs_read_super()`

This method fills in the fields of the private data structure by parsing the mount options. Figure 4.1 shows the fields in the Simple XCacheFS's private data in the super block object.

Primarily the information global to the file system is stored in this object. It holds pointers to the lower file systems' superblock objects of SB and CB. In addition to this, certain metadata fields that are determined during the mount and are constant during throughout a mounted instance of Simple XCacheFS are maintained in this private structure. The field `pseudo_cb_dentry` is a reference to dentry for the pseudo directory representing CB itself. As discussed in the design of encoding unique filenames, the unique identifier representing the absolute path of SB's root, is written to a file and is

stored in the `root_hash` field of this private structure.

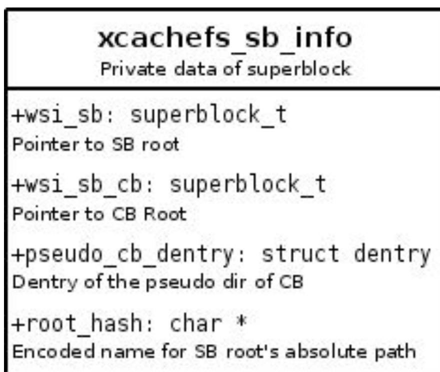

Figure 4.1: Private data of Simple XcacheFS super block

4.2. Encoding of unique file names

The absolute path name of the file/directory in the Source Branch is used to encode unique filenames in Cache Branch. The pathname of SB's root is hashed to obtain a unique identifier for the SB. This is used to further generate unique file names for the files that are cached in CB. Essentially, instead of determining the file/directories absolute path and hashing it, the unique file name for any file is obtained by appending the file name to its parent's (pseudo directory in CB) unique file name and hashing it.

To enable encoding of paths to unique file names in CB even when SB is not available, the SB's root directory's encoded path is stored as persistent data in a file in CB. The routine `xcachefs_get_ encoded_name` encodes a file name in SB to unique filename. This routine uses encoded name of the parent directory (can be root also) and generates a new encoded name using MD5 hashing algorithm.

Given a path name `/xcachefs/sb` of SB's root directory, the unique identifier representing SB is generated and stored persistently in the file *.root_hash* in CB. When a file in SB, say `/xcachefs/sb/file.txt` is accessed, the name `file.txt` is appended to its parent's unique identifier (in this case, the unique identifier for SB i.e. the `root_hash`, since the file exists under SB).

As an example, consider the following:-

Absolute path name of SB's Root:
`/xcachefs/sb.`
Unique identifier for SB's root (using MD5):
`001dc49646d280c53a6c1d2f961203ed`
File name in SB:
`file.txt`
Unique identifier for file `file.txt` before hashing:
`001dc49646d280c53a6c1d2f961203edfile.txt`
Unique file name of `file.txt` in CB:
`10191b843e27d20bb56d8e508d3c6879`

4.3. File/Directory Lookup

In Simple XCacheFS, a lookup on a file or a directory in SB results in caching of all the directories encountered in its path.

When SB is available, a lookup is performed in SB. If the file/directory exists in SB, a lookup is performed with its encoded name in CB. If the file exists in CB, then a success is returned, else the file is created in CB with this encoded name. Once created, the parent pseudo directory is updated with the dirent of the newly created file or directory. When SB is offline, a lookup is performed for the encoded filename in CB.

The type of the pseudo directory is to be maintained as `S_IFDIR` in xcachefs level inode even if the underlying CB's inode's type is a regular file. Hence, during a lookup, if SB is available, the type is read from the SB's inode's `i_mode` field. If SB is offline the type of an inode is obtained by reading the type from the dirent stored in the parent pseudo directory.

Method: `xcachefs_lookup()`
This method implements the lookup operation as described above. Following is a simple pseudo code to show the sequence flow of this method.

```
XCACHEFS_LOOKUP(dentry)

if ( SB AVAILABLE )
   lookup(dentry->d_name, Parent directory in SB)
   if FAILURE
      return NEG_DENTRY
      else
  lookup(name(dentry->d_name), CB)
  if NEG_DENTRY
     Create file/dir in CB
     Update pseudo directory with the new dirent
     Obtain type of inode from Lower_Sb inode.
```

Update the type in xcachefs level inode.
return SUCCESS

if (SB OFFLINE)
lookup(encode_name(dentry->d_name), CB)
if FAILURE
return NEG_DENTRY
else
Obtain the type of inode from parent pseudo dir.
Update the type in xcachefs inode.
return SUCCESS

4.4. Reading and Listing of directory contents

Reading of directories is straight forward when Source Branch is available. A `readdir` operation on Simple XcacheFS is simply redirected to the lower SB's file system since the underlying inode is of the type Directory.

When SB is not available, accessing and modifying of files in CB is still allowed. Since the underlying inode in the CB is of Regular File type, the `readdir` operation on Simple XcacheFS cannot be redirected to the CB file system. Hence, for a `readdir` operation on Simple XCacheFS when SB is offline, the pseudo directory is opened and the dirents are read. The fields in the dirents are passed to `filldir` which will fill the user buffer for `readdir` with appropriate content. To accomplish this, certain helper routines `xcachefs_get_dirent`, `xcachefs_get_dirent_pos` are implemented. These routines read the pseudo directory and return content and position of the dirents.

5. EVALUATION

5.1. Performance evaluation:

To evaluate the performance of Simple XcacheFS, **lat_fs** test from LMBENCH [3] is run on a directory with Simple XCacheFS mounted on it. The same test is performed for xcachefs which caches files in the traditional way (copies the directory hierarchy).

Lat_fs is a program that creates a number of small files in the specified directory and then removes the files. Both the creation and removal of the files is timed. Tables 5.1 and 5.2 show the results obtained for this test and Figures 5.1 and 5.2 show the comparison of the latencies. It can be observed that the latency has increased on Simple XCacheFS. This overhead can be attributed to the additional encoding and decoding that is required for having a flat hierarchy.

File Size	Number Created	Creations per Sec	Removals per Sec
0 kb	1000	84	79
1 kb	1000	47	57
4 kb	1000	55	55
10 kb	1000	41	55

Table 5.1: Results obtained for lat_fs (LMBENCH) test for Simple XCacheFS. (Flat Hierarchy Cache Stackable File System)

File Size	Number Created	Creations per Sec	Removals per Sec
0 kb	1000	139	58
1 kb	1000	71	57
4 kb	1000	71	77
10 kb	1000	62	79

Table 5.2: Results obtained for lat_fs (LMBENCH) test for XCacheFS. (Traditional Stackable Caching File system)

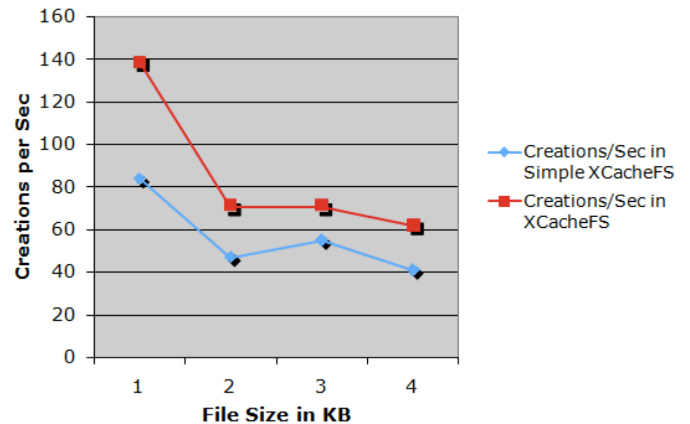

Figure 5.1: Comparison of number of creations per second, on Simple XCacheFS and XCacheFS.

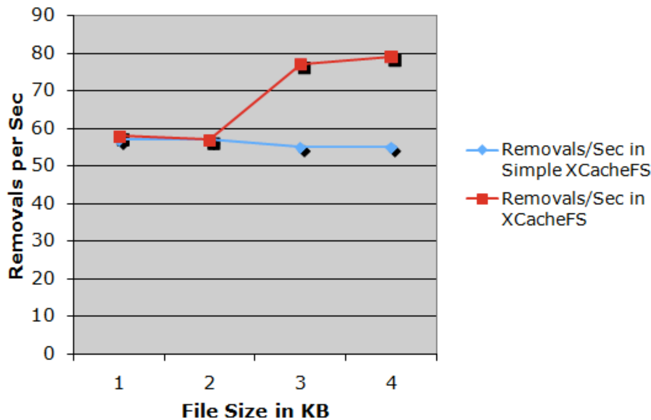

Figure 5.2: Comparison of number of removals per second, on Simple XCacheFS and XCacheFS.

5.2. Functional Testing

Simple XCacheFS is tested for basic file system operations and functionality specific to Flat Caches. The tests performed and the corresponding results are listed in Appendix A.

6. FUTURE WORK

To bring out the advantage of having a flat hierarchy of cache, it would be useful to implement and evaluate the performance of some of its applications. One future work is to implement eviction and compare the performance with traditional caching file system.

Since updates to the cache are allowed even when SB is offline and there could be offline changes to SB, there should be some way to synchronize the content on either or both of these branches. Synchronizing the content would require some policy to be implemented. One policy would be to provide only one way synchronization either from SB to CB or from CB to SB. Another policy would be to synchronize the content on both the branches. Supporting bidirectional synchronization would require additional logging for each file that is modified (time attributes before and after modification). This would add additional overhead to the file system.

7. CONCLUSIONS

Simple XcacheFS successfully implements a stackable flat caching file system. It should be noted that Simple XCacheFS is a basic implementation on top of which useful features can be added to bring out the advantages of a flat caching file system.

8. REFERENCES

[1] Erez Zadok and Jason Nieh. FiST: A Language for Stackable File Systems. *Proceedings of 2000 USENIX Annual Technical Conference.*

[2] Gopalan Sivathanu and Erez Zadok. A Versatile Persistent Caching Framework for File Systems, *Stony Brook University, Technical Report FSL- 05-05.*

[3] LMBench - Tools for Performance Analysis. *http://lmbench.sourceforge.net/*

APPENDIX A

Mounting when SB is available:

```
# mount -t xcachefs -o sb=/n/fist/xcachefs/sb,
cb=/n/fist/xcachefs/cb none /mnt/xcachefs
```

Mounting when SB is not available (offline):

```
# mount -t xcachefs -o sb=NA,cb=/n/fist/xcachefs/cb none
/mnt/xcachefs
```

Creating and deleting files and directories:

When SB is available, if files/directories are created/deleted, they are created/deleted both from SB and CB. When SB is not available (offline), if files/directories are created/deleted, they are created/deleted only in CB.

i. When SB is available:

```
# ls /n/fist/xcachefs/sb
# ls /n/fist/xcachefs/cb
```

// Create file creates a file in both SB and CB.

```
# ps > /mnt/xcachefs/file1
# ls /n/fist/xcachefs/sb
file1
# ls /n/fist/xcachefs/cb
00e61bbb77fa2eb146bb2b35bcfb4912
```

//mkdir creates directory in both SB and CB.

```
# mkdir /mnt/xcachefs/dir1
# ls /n/fist/xcachefs/sb
dir1  file1
# ls /n/fist/xcachefs/cb
00e61bbb77fa2eb146bb2b35bcfb4912
6594a479e53938ec09cfed89d4ed85a2
```

//rmdir deletes directory in both SB and CB.

```
# rmdir /mnt/xcachefs/dir1
# ls /n/fist/xcachefs/sb
file1
# ls /n/fist/xcachefs/cb
00e61bbb77fa2eb146bb2b35bcfb4912
```

ii. When SB is not available (offline):

```
# ls /n/fist/xcachefs/sb
dir1  file1
# ls /n/fist/xcachefs/cb
00e61bbb77fa2eb146bb2b35bcfb4912
6594a479e53938ec09cfed89d4ed85a2
```

//The directory will be deleted from CB, but not from SB.

```
# rmdir /mnt/xcachefs/dir1
# ls /n/fist/xcachefs/sb
dir1  file1
# ls /n/fist/xcachefs/cb
00e61bbb77fa2eb146bb2b35bcfb4912
```

//mkdir creates directory in CB, but not in SB.

```
# mkdir /mnt/xcachefs/dir2
# ls /n/fist/xcachefs/sb
dir1  file1
# ls /n/fist/xcachefs/cb
00e61bbb77fa2eb146bb2b35bcfb4912
e55d1fbeb47a788cdf65389d4cb89aae
```

Listing of directory content:

When SB is available, ls is performed on SB. When SB is not available (offline), ls is performed on CB and lists files present in the directory (in CB) on which ls is done.

i. When SB is available:

//Contents of dir1 in SB.

```
# ls /n/fist/xcachefs/sb/dir1
file1  file2  indir1
```

//ls on dir1 shows the files and directories in dir1 of SB when Simple XCacheFS is mounted.

```
# ls /mnt/xcachefs/dir1
file1  file2  indir1
```

//On removing indir1

```
# rmdir /mnt/xcachefs/dir1/indir1
# ls /mnt/xcachefs/dir1
file1  file2
# ls /n/fist/xcachefs/sb/dir1
file1  file2
```

ii. When SB is not available (offline):

//Contents of dir1 in SB.

```
# ls /n/fist/xcachefs/sb/dir1
file1  file2  indir1
# ls /mnt/xcachefs/dir1
file1  file2  indir1
```

//On removing indir1, it gets removed from CB, but not from SB

```
# rmdir /mnt/xcachefs/dir1/indir1
# ls /mnt/xcachefs/dir1
file1  file2
# ls /n/fist/xcachefs/sb/dir1
file1  file2  indir1
```

Renaming files or directories:

When SB is available, if a file/directory is renamed, the file/directory will be renamed both in SB and CB. When SB is not available (offline), if a file/directory is renamed, the file/directory will be renamed only in CB.

i. When SB is available:

```
# ls /n/fist/xcachefs/sb/dir1
file1  file2  indir1
# ls /mnt/xcachefs/dir1
file1  file2  indir1
```

//Renames in both SB and CB

```
#mv  /mnt/xcachefs/dir1/indir1  /mnt/xcach
efs/dir1/indir2
# ls /mnt/xcachefs/dir1
file1  file2  indir2
# ls /n/fist/xcachefs/sb/dir1
file1  file2  indir2
```

ii. When SB is not available (offline):

```
# ls /n/fist/xcachefs/sb/dir1
file1  file2  indir1
# ls /mnt/xcachefs/dir1
file1  file2  indir1
#mv /mnt/xcachefs/dir1/indir1 /mnt/xcachefs
w/dir1/indir2
# ls /mnt/xcachefs/dir1
file1  file2  indir2
# ls /n/fist/xcachefs/sb/dir1
file1  file2  indir1
```

Caching of directories as files (pseudo directories) in CB:

```
# ls /n/fist/xcachefs/sb
# ls /n/fist/xcachefs/cb
# mkdir /mnt/xcachefs/dir1
```

//directory in sb

```
# ls /n/fist/xcachefs/sb
dir1
```

//corresponding pseudo file in cb

```
# ls /n/fist/xcachefs/cb
6594a479e53938ec09cfed89d4ed85a2
```

//stat on sb's directory

```
# stat /n/fist/xcachefs/sb/dir1
  File: `/n/fist/xcachefs/sb/dir1'
  Size:  4096                   Blocks:  8
IO Block: 4096   directory
```

//stat on directory through xcachefs

```
# stat /mnt/xcachefs/dir1
  File: `/mnt/xcachefs/dir1'
  Size:  0                      Blocks:  0
IO Block: 4096   directory
```

```
#stat on pseudo directory in cb
/n/fist/xcachefs/cb/6594a479e53938ec09cfed
89d4ed85a2
File:
`/n/fist/xcachefs/cb/6594a479e53938ec09cfe
d89d4ed85a2'
Size: 0  Blocks: 0  IO Block: 4096
regular empty file
```