Implement FS-Cache support in FUSE kernel module

PROJECT REPORT
Under the guidance of

Niels De Vos,
Senior Software Engineer,
Red Hat Inc
Amsterdam, Netherlands

Submitted by
Vimal Kumar A.R
511237799

in partial fulfillment of the requirement for the award of the degree of

Bachelors of Computer Applications

Department of Information Technology

Sikkim Manipal University
Directorate of Distance Education

December 2014

Sikkim Manipal University

Directorate of Distance Education

BONAFIDE CERTIFICATE

Certified that this project report titled “Implement FS-Cache support in FUSE kernel module” is the bonafide work of Vimal Kumar A.R (Reg number 511237799), who carried out the project work under my supervision.

Signature:	Signature:
Head of the LC	Guide
Name:	Name:
Address:	Address:

Abstract

This dissertation discusses the effort to enable support for the FS-Cache caching mechanism in FUSE (File system in User space) module in Linux kernel. We aim to get a working prototype rather than a full fledged caching implementation, since the latter would not be suitable or possible for a 3 to 4 month under-graduate project.

The framework consists of the FS-Cache utility, its back-end caching mechanism 'Cachefilesd', the GlusterFS storage cluster, and the FUSE kernel module by which the GlusterFS cluster gets accessed on a client machine.

This dissertation talks about the most essential concepts such as how FUSE works in general, an over view of GlusterFS, how FS-Cache support in FUSE may help in reducing the data access times for a GlusterFS client over the network in certain conditions etc. We then look into how the FS-Cache API can be used to implement the feature.

As said above, this is an initial prototype, and further changes are expected in this feature to tweak the behavior as well as add features, depending on feedback from the upstream Linux kernel community which includes developers and end-users.

This project work is done as a feature implementation for the upstream GlusterFS project, and is covered under the GNU General Public License (GPL) [1]. This and any additional improvements to this feature will be considered open and accessible to the public, as per GPL terms and conditions.

Table of contents

i. Abstract iv
ii. Table of contents v
iii. List of figures vii

1. Introduction
1.1. Background 9
1.2. Motivation 9

2. Requirement specification
2.1. Linux kernel source 11
2.2. The operating system – Fedora 20 11
2.3. Git (version control system) 11
2.4. Cscope 12
2.5. Prepare the setup 12
2.5.1. Install the virtual machines 12
2.5.2. The kernel source rpm, from Fedora 13
2.5.3. Prepare the kernel source 13
2.5.4. Track the kernel source with git 14
2.5.5. Build the cscope database 14
2.5.6. Compile the 'fuse' module 15
2.5.7. Load the module 15

3. Design, and the components
3.1. Introduction 17
3.2. FUSE (File system in User space) 17
3.2.1. Introduction 17
3.2.2. Benefits of using FUSE 17
3.2.3. How does FUSE work? 18
3.2.3.1. A general description 18
3.2.3.2. An in-depth look into fuse internals . 19
3.3. GlusterFS
3.3.1. Introduction 20
3.3.2. Benefits of GlusterFS 21
3.3.3. Attributes of GlusterFS 21
3.4. FS-Cache
3.4.1. Introduction 22
3.4.2. Different methods of back-end caching .. 23
3.4.2.1. CacheFS 23
3.4.2.2. Cachefilesd 24
3.4.3. The back-end caching structure for cachefilesd 24
3.4.3.1. Indexes and Objects 25

3.4.4. An example showing how netfs data is cached 26

4. Implementation: Integrating FS-Cache in FUSE

4.1. The FS-Cache netfs API 28
4.2. Cookies 28
4.3. Register and Unregister the with the netfs API 29
4.4. Important data-structures in various components 30
4.4.1. fuse_inode 30
4.4.2. fuse_conn 31
4.4.3. fscache_cookie_def 31
4.5. Create the indexes 32
4.5.1. 1st level index - 'struct fuse_cache_netfs' 32
4.5.2. 2nd level index - 'struct fuse_subfs_cache_index_def' 32
4.5.3. 3rd level index - 'struct fuse_mnt_cache_index_def' 33
4.5.4. 4th level index - 'struct fuse_fh_cache_index_def' 33
4.6. Functions defined in each index 34
4.6.1. 1st level index - 'fuse_cache_netfs' 34
4.6.2. 2nd level index – 'fuse_subfs_cache_index_def' ... 34
4.6.3. 3rd level index – 'fuse_mnt_cache_index_def' 35
4.6.4. 4th level index – 'fuse_fh_cache_index_def' 36
4.7. Create cookies for each index levels. 37
4.8. Fill the indexes with the cache 37
4.9. Return cached data from the local storage. 38
4.10. The code for the working prototype 39
4.10.1. A git one-liner list of the patches 39
4.10.2. An overview of the total changes 40

5. Testing

5.1. Scenario 41
5.2. Prepare the Gluster cluster nodes 41
5.2.1. Install the GlusterFS nodes. 41
5.2.2. Create the back-end bricks 42
5.2.3. Install the glusterfs server packages 43
5.2.4. The glusterd process 44
5.2.5. Create a trusted pool, or peer group. 44
5.2.6. Create a volume 44
5.2.7. Start the volume 45
5.3. Prepare the client machine 45
5.3.1. Install the glusterfs client packages 46
5.4. Build, load, and check the 'fuse' kernel module 46
5.5. Configure FS-Cache to use cachefilesd 47
5.6. Mount the glusterfs volume 48
5.7. Write/Read data to the glusterfs mount point 49
5.8. Check the data cached on the local storage 52

6. Limitations, and future improvements 53

7. Conclusion 54

8. References 57

List of figures

3.1. fuse - libfuse interaction 18
3.2. GlusterFS architecture 20
3.3. FSCache interaction between the netfs and back-end cache 23
3.4. FSCache and Cachefiles interaction 24
3.5. Cachefilesd backend structure 27

5.1. The Fedora 20 boot screen 41
5.2. The login prompt 42
5.3. Adding a virtual hard-disk to the existing Fedora installation 43
5.4. /proc/fs/fscache/objects 51
5.5. Listing the cache contents at /var/cache/fscache/ 52

Chapter 1 - Introduction

In this chapter, we look at the background and motivation behind implementing support for FSCache in FUSE. This feature will help in caching data for file systems which work over FUSE. We will inspect this specifically for GlusterFS, an open source software storage solution that works over FUSE. We will also discuss the scope, and application of this feature in GlusterFS.

1.1. Background

Caching is the process of transparently storing data on local storage, accessed from a primary media such as the network, (or a locally mounted CD/DVD-ROM) so that future requests can be served faster from the cache rather than accessing the primary media. The concept of caches has been prevalent for quite some time. The most common example on caching are the various levels of processor caches (L1, L2, etc..). The data presiding in L1/L2 cache memory can be accessed much faster compared to fetching it from the external hard disk.

Another example is the kernel page cache, where frequently read/accessed pages are kept in memory for a longer time so that further requests are served faster.

1.2. Motivation

There are several file system implementations using the FUSE kernel module. The FUSE module helps developers write file system implementations without venturing into kernel layer. It helps developers avoid the intricacies of kernel programming, and the long and detailed procedures of getting a file system driver included the kernel. In many cases, file systems created for a specific purpose would find it difficult to be included in the upstream Linux kernel project. It may very well go as a user space application working over FUSE. This also makes it much easier to port, since it depends only on FUSE.

Several of these file system communities are interested in a dedicated caching feature. This project work is intended to create a working prototype for a cache implementation for the GlusterFS file system, which can then be submitted to the upstream communities for further evaluation and comments. Further suggestions and testing by the community will pave way for the finalized version, and may get included in the Linux mainstream kernel.

This dissertation discusses a method of caching the data using FSCache, specifically for GlusterFS. We will look into how this can be used to increase read speeds on a GlusterFS client, which works via the FUSE kernel module to read data over the network from a GlusterFS cluster.

This implementation would help in certain workloads where the access mostly consists of reads, and the network speeds are low. The difference may be considerable, in such cases. Initial reads should cache the data on the client machine under a predetermined location, and subsequent reads would not need to cross the network to fetch the data, unless it has changed, which ultimately increases the read speeds.

Chapter 2 - Requirement specification

2.1 Linux kernel source

Linux [2] is a free, open source [3], and POSIX (Portable Operating System Interface) [4] compliant operating system. Linux, specifically is the term for the kernel, on top of which other open source applications operate, and together form the operating system.

The Linux kernel was created by Linus Torvalds in 1991 as a pet project, and it has grown considerably ever since. As of now, it is a preferred platform for servers, main-frame computers, and super computers. 97% of the top 500 super computers [5] use the Linux kernel, while the rest 3% uses a variant of Unix. The reach of Linux is growing rapidly due to popularity of Android devices as well as other portable devices such as Raspberry Pie, and other hand-held devices.

We will be using the Linux kernel source to work this project. The code would be an extension to the fuse code in the Linux kernel. But since the Linux kernel is a rapidly changing code base, we limit our work to a specific kernel version, ie.. 3.16.6-200.

We will use the kernel source from the Fedora project, which comes in a packaged form named RPM [6]. This makes it easy to compile, load, and test the FUSE module on a Fedora virtual machine.

The specific kernel version can be downloaded from [7] as a source rpm.

2.2 The operating system – Fedora 20

Fedora [8] is community based operating system based on the Linux kernel, and is a completely free version among the various Linux flavors available today.

Three Fedora 20 virtual machines will be used, two which acts the glusterfs cluster, while the third acts as the client. The code will be compiled and tested on the third node.

All three Fedora 20 machines have 10GB of hard disk space, and 1GB of memory. The GlusterFS cluster nodes will have one 1GB disk additional, to act as its GlusterFS backend storage. The virtual machines are running on the KVM [9] hypervisor, which is used to manage the virtual machine and its resources. We will be using the 'virt-manager' [10] interface to install and boot the virtual machine, assign storage if needed etc..

2.3 Git (version control system)

'Git' [11] is a free and open source version control system, that can be used for small and large projects equally. It is fast and efficient compared to the other version control

systems such as cvs, svn etc..

Git was designed and developed by Linus Torvalds for Linux kernel development. We will be using it for the following :

a) Create a local repo including the linux kernel source.
b) Create branches in the kernel source to isolate the code we write.
c) Commit the added/edited code in regular intervals.
d) Create patches from the code changes.
e) Apply patches to our code branch.
f) List and recall the patches we have committed.

The git commands used, will be demonstrated in the coming sections as required. To install git, we will use 'yum' [12].

 # sudo yum install git -y

Once installed, the command 'git' would be available for use, along with its sub-commands.

2.4. Cscope

Cscope [13] is a text-based interface for browsing C source code. It can be used to search functions, declarations, definitions, strings etc.. inside a C source-code directory.

Cscope is installed on the virtual machine using yum.

 # sudo yum install cscope -y

2.5. Preparing the setup.

2.5.1. Install the virtual machines

We will install three virtual machines, two acts as GlusterFS cluster nodes, while the third node acts as the client where the caching will be happening. All the machines will have 10GB of disk space, and 1GB of memory.

The virtual machines are installed on top of a Fedora 21 host machine. The operating system (Fedora 20) ISO is downloaded from [13] and installed as per the installation documentation [15], using 'virt-manager'.

NOTE: The virtual machines have a user account named 'test' which has 'sudo' access configured. We normally prefer to work as the 'test' user. In need of 'root' privileges, use 'sudo' to get privileged access.

2.5.2 The kernel source rpm, from Fedora

Download the kernel source rpm from the Fedora repository either through the browser or a terminal utility such as 'wget' or 'curl'. We will be using 'wget' in this case.

 # wget http://kojipkgs.fedoraproject.org/packages/kernel/3.16.6/200.fc20/src/kernel-3.16.6-200.fc20.src.rpm

2.5.3 Prepare the kernel source

a) Once the kernel source is downloaded as per the step in section 2.5.2, install the package using the 'RPM package manager'. This will create a folder named 'rpmbuild' in the user's home directory, with further sub folders.

 # rpm -ivh kernel-3.16.6-200.fc20.src.rpm

b) Install the 'kernel-devel' rpm using 'yum'. This package provides the the kernel header files as well as Makefiles, required for building modules for that specific kernel version. The files should be in /usr/share/kernels/.

 # yum install kernel-devel-3.16.6-200.fc20

c) Build an rpm package from the spec file in the kernel source package we installed in step (a), For more details on 'rpmbuild', see the man page (*man rpmbuild).*

 # cd ~/rpmbuild
 # rpmbuild -bp ~/rpmbuild/SPECS/kernel.spec

d) The prepared source should be in ~/rpmbuild/BUILD/kernel-<version>/linux-<version>/. Move it to the folder /Kernel-source.

 # mkdir /Kernel-source
 # mv ~/rpmbuild/BUILD/kernel-3.16.fc20 /Kernel-source

e) Change the working directory to '*~/rpmbuild/BUILD/kernel-3.16.fc20 /Kernel-source',* and copy the existing *'/boot/config-3.16.6-200.f20.x86_64'* file to the current folder.

 # cd ~/rpmbuild/BUILD/kernel-3.16.fc20 /Kernel-source
 # cp /boot/config-3.16.6-200.f20.x86_64 .config

f) Enable the following keys in '.config' so that these are enabled when the code is compiled.

 # echo CONFIG_FUSE_FSCACHE=y >> .config
 # echo CONFIG_FSCACHE_HISTOGRAM=y >> .config
 # echo CONFIG_FSCACHE_STATS=y >> .config

g) The above settings will enable the following:

a) CONFIG_FUSE_FSCACHE - Enable the FUSE-FSCACHE support in kernel.
b) CONFIG_FSCACHE_HISTOGRAM - Enable the FSCache histogram logging
c) CONFIG_FSCACHE_STATS - Enable the statistics for FSCache

h) Run 'make oldconfig' to read the copied '.config' file and prompt the user for un-configured options in the current kernel source, which are not found in the .config file.

```
# make oldconfig
```

This prepares the kernel source for our code and build requirements.

2.5.4. Track the kernel source with git

We will add/edit source-code inside /fs/fuse/. In order to keep track of the changes, as well as manage the patches, git will be used to monitor the source files.

a) Change the working directory to */Kernel-source/kernel-3.16.fc20/linux-3.16.6-200.fc20.x86_64/*

```
# cd /Kernel-source/kernel-3.16.fc20/linux-3.16.6-200.fc20.x86_64/
```

b) Create a git repository locally.

```
# git init .
```

c) Mark the contents in /fs/fuse/ to be monitored by git.

```
# git add fs/fuse/
```

d) Create a git branch [16] to work on a separate branch other than 'master'.

```
# git checkout -t -b wip-fuse-fscache origin/master
```

e) Commits are done regularly to keep track of the code additions.

```
# git commit -am "<Commit message>"
```

The commit message should be meaningful, with a description of the change.

2.5.5 Build the Cscope database.

We will use cscope to search the functions, symbols etc.. A cscope database has to be built prior staring to use this. This database is referred by cscope, and hence speeds up the process considerably.

a) Build the cscope database by recursively searching through the directories:

 # cscope -Rv

b) Start using the database with

 # cscope -d

2.5.6. Compile the fuse module

The fuse source code is compiled as following:

 # make -C /lib/modules/$(uname -r)/build M=$PWD/fs/fuse/ clean

 # make CONFIG_FUSE_FSCACHE=y EXTRA_CFLAGS=-DCONFIG_FUSE_FSCACHE -C /lib/modules/$(uname -r)/build M=$PWD/fs/fuse/

2.5.7. Load the fuse module

The 'fuse' module has dependencies on 'fscache', and hence 'fscache' should be loaded prior 'fuse'.

 # modprobe fscache
 # insmod fs/fuse/fuse.ko

Use 'lsmod' to confirm if the module is loaded properly.

 # lsmod | grep fuse

In the next chapter, we will look at the components which are the vital part of this project, ie.. FSCache, FUSE, and GlusterFS.

Chapter 3 - The components, and their design

3.1 Introduction

We will talk about the three main components for this feature implementation, ie.. FUSE, GlusterFS, and FSCache. We would also present a generic design of the components, its working internals, and relation with other components.

3.2. FUSE (File system in Userspace)

3.2.1. Introduction

File system in userspace (FUSE) is a mechanism in Unix-like operating systems that enables non-privileged users to create their own file systems, without adding code in the main kernel tree.

FUSE is a user-space file system framework. It consists of a kernel module (fuse.ko), a user-space library (libfuse.ko), and a mount utility (fusermount). The most important feature of FUSE is allowing secure, and non-privileged mounts. The 'fuse.ko' kernel module provides a bridge from the file system code running in user-space, to the kernel space.

Before FUSE, file system programming required extensive knowledge of the kernel internals, as well as VFS[17], the Virtual File System. A high level knowledge of the C programming language and debugging was needed as well.

With the invent of FUSE, user-space file systems can be written in various languages including interpreted languages such as Python, Ruby etc.. This helps developers not worry about the kernel internals and C programming/debugging.

A file system working via FUSE is different from traditional file systems in the kernel, mainly because the it does not store the data, but will pass it onto an underlying file system (such as ext4, XFS etc..) via the kernel interfaces.

There are a wide variety of user-space file systems developed over FUSE and a list can be seen at [18].

3.2.2. Benefits of using FUSE

There are plethora of benefits while writing file systems using the FUSE interface.

a) Portable file systems are possible due to the fact that it interacts with the fuse module, rather than the kernel directly. Hence file systems written for Linux, may also be fit for

other versions of UNIX.

b) Write a file system in other than C programming language.

c) Increased system security since the file system is running as a separate process outside the kernel.

d) A file system crash does not crash the kernel and does not bring the machine down.

e) A hang in the file system code can be killed by passing a 'kill' signal.

f) A file system running over fuse is like a normal process and hence can be resource controlled using Cgroups, or other resource-limiting utilities.

3.2.3 How does FUSE work?

3.2.3.1. Generic description

As said in section 3.2.1, the FUSE framework consists of a kernel module named 'fuse.ko', a user-space library 'libfuse', and a mount utility named 'fusermount'.

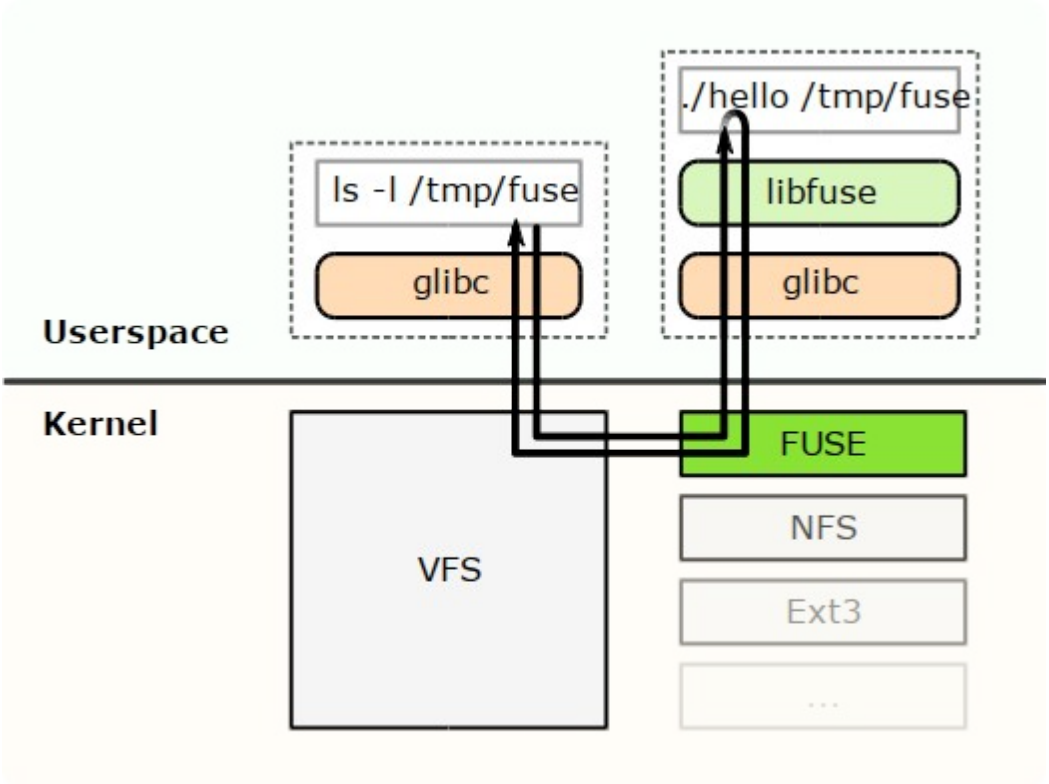

Fig 3.1 – fuse – libfuse interaction

A more detailed look into how a user-space file system works over the FUSE framework:

a) A user-space file system requests common file operations such as 'open', 'read', 'write', 'close' etc..

b) glibc passes these system calls to the VFS layer which acts as a common interface to talk with underlying file system drivers.

c) The VFS layer directs these to the 'fuse.ko' module loaded in memory.

d) The 'fuse.ko' module needs to communicate with the FUSE library 'libfuse' to execute the operations and this needs /dev/fuse to be opened. Hence, 'fuse.ko' goes to open /dev/fuse through 'glibc' and uses the file descriptor obtained to communicate with 'libfuse'. See 'NOTE' below.

f) The end-operations which include actual read/write from the underlying on-disk file system, goes through VFS onto the respective file system driver such as 'EXT4', 'XFS' etc..

NOTE: The fuse.ko kernel module and the 'libfuse' library communicate via a file descriptor which is created by opening /dev/fuse. It can be opened multiple times and hence multiple file systems working via FUSE can be mounted.

3.2.3.2. An in-depth look into fuse internals

a) When a user process access a file over fuse, the 'fuse_main()' function (lib/helper.c) parses the arguments passed on by the user-space program, and in-turn calls 'fuse_mount()' (lib/mount.c).

b) The 'fuser_mount()' (lib/mount.c) function creates a unix domain socket, then forks and executes 'fusermount' (util/fusermount.c) passing it one end of the socket. Since the 'mount()' system call is a privileged operation and FUSE supports non-privileged mounts, the helper program 'fusermount' is called, which is a setuid root program, to mount the file system.

c) 'fusermount()' (util/fusermount.c) ensure that the 'fuse.ko' module is loaded in memory since it needs to open '/dev/fuse'. Once 'fuse.ko' is loaded, it opens /dev/fuse and passes the file-descriptor back to 'fuse_mount()' (lib/mount.c)
d) 'fuse_mount()' returns the file-descriptor from /dev/fuse, to 'fuse_main()'.

e) Once the /dev/fuse file-descriptor is returned to 'fuse_main()', 'fuse_main()' calls 'fuse_new' (lib/fuse.c) which happens to be a pointer to the data-structure struct 'fuse' (lib/fuse.c) and hence allocates a new memory segment which stores and maintains a cached image of the file system data.

f) After the 'fuse_new' data structure has been allocated, 'fuse_main()' will call either 'fuse_loop()' (lib/fuse.c) or 'fuse_loop_mt()' (lib/fuse_mt.c). These start to read the file system calls from /dev/fuse which in turn call the usermode functions in the struct 'fuse_operations' (include/fuse.h), and lastly calls 'fuse_main()'.

g) The results of the calls from 'fuse_loop()' and 'fuse_loop_mt()' to the struct 'fuse_operations' are written back to /dev/fuse file, which delivers it back to the system calls.

3.3. GlusterFS

3.3.1. Introduction

GlusterFS [19] is an open source, distributed file system capable of scaling to several peta-bytes (1PB=1000 TB), and can handle thousands of clients.

A GlusterFS cluster pools together storage from several nodes over the network, via TCP/IP or RDMA. It aggregates disk and memory resources and manages the data in a single name-space.

GlusterFS supports standard client-server protocols such as NFS, CIFS, and its very own glusterfs client implementation which runs over FUSE.

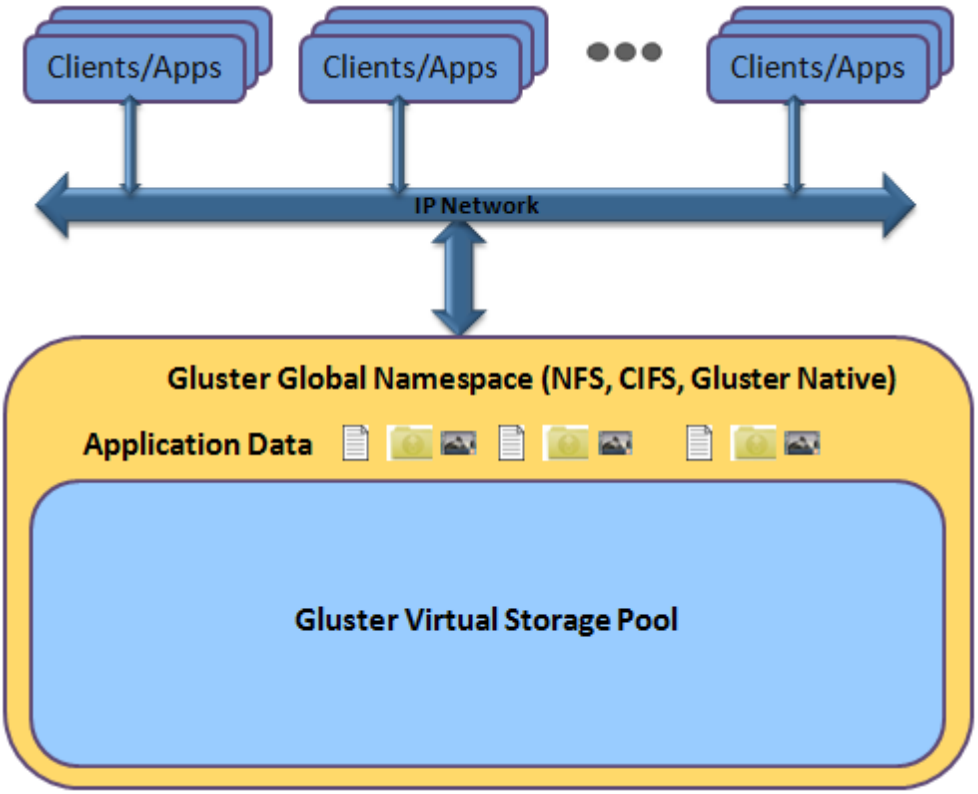

Fig 3.2. GlusterFS architecture

The above diagram Fig 3.2. shows a bird's eye view of GlusterFS architecture.

3.3.2. Benefits of GlusterFS over traditional storage solutions

There are several benefits for GlusterFS over the traditional storage implementations. A few of them are:

a) GlusterFS supports common network protocols such as NFS, CIFS, and a native client implementation which runs over FUSE on the client.

b) Traditional TCP/IP networks can be used, whereas traditional SAN storage use Fibre Channels for communication which are much costly.

c) GlusterFS has the ability to scale-out on a large scale to several peta-bytes with commodity hardware at a minimal cost compared to the normal SAN storage which are almost always monolithic.

d) GlusterFS is capable of handling a large number of clients, over multiple protocols.

e) GlusterFS is compatible with all the major operating systems such as Microsoft Windows, Apple Mac OS, Linux, and various Unix flavors such as BSD, Solaris, etc..,

f) GlusterFS uses an Elastic Hashing Algorithm to locate files in the back-end storage. This avoids the need of a centralized metadata server, and rules out the bottle neck of the centralized meta data server going down and restricting access. This architecture reduces the I/O hops and ensures better performance, linear scalability, and reliability.

3.3.3. Attributes of GlusterFS

The following can be seen as the attributes of the GlusterFS storage solution.

a) Scalability and Performance.

b) High availability

c) Global namespace

d) Elastic Hashing Algorithm

e) Elastic Volume Manager

f) Standards based (TCP/IP, RDMA, NFS, CIFS, etc..)

We will discuss on how to setup a GlusterFS server, how to form a cluster, how to create a volume, and how to access it, in Chapter 5.

3.4. FS-Cache

3.4.1. Introduction

FS-Cache [20] is a persistent local caching mechanism that can be used to cache data retrieved from the network, on the local disk. This helps minimize the data traffic over the network, mainly for data that remains unchanged, and can also help speed up client data access for network file systems such as NFS, CIFS, AFS, etc.. when the network speed is on the low side. This mechanism can cache whole files or parts of files on local storage.

FS-Cache is a thin layer between the file system and the underlying caching mechanism, and strives to be transparent to the users and administrator. It does not alter the basic operation of a network file system.

FS-Cache needs a caching back-end to cache the data locally. A cache back-end is a storage driver configured for caching. FS-Cache uses extended attributes (xattrs) and it needs a file system which supports xattrs. Almost all the recent file systems support extended attributes, such as EXT3, EXT4, XFS etc..

The following diagram Fig 3.3, shows a generic overview on how FS-Cache fits in between network file systems and the back-end storage mechanism.

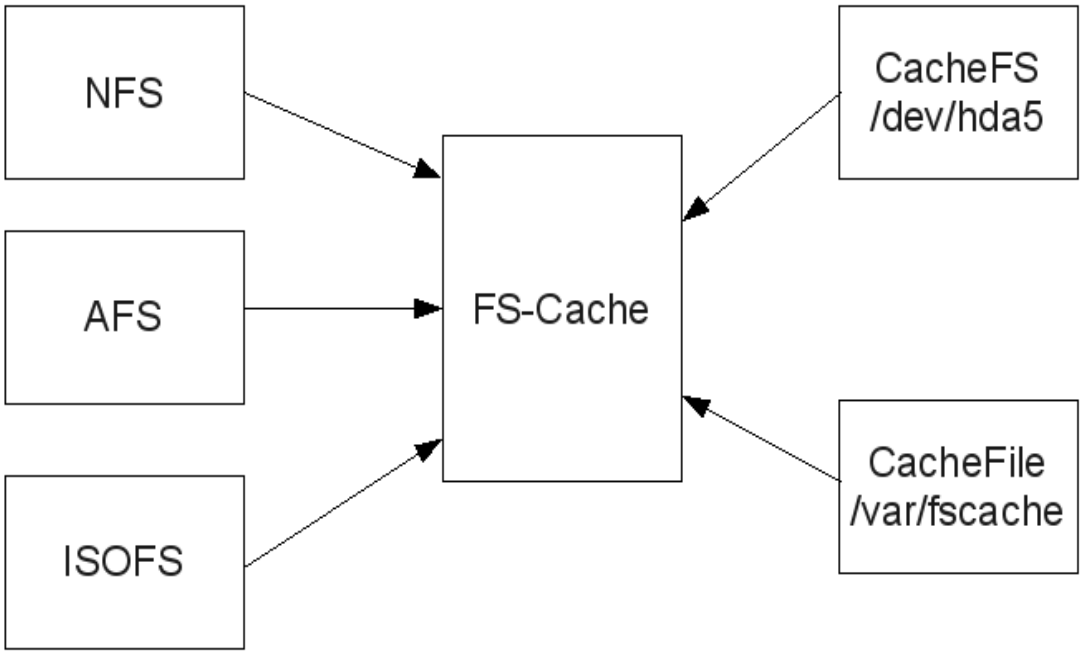

Fig 3.3 – FS-Cache interaction between the netfs and back-end cache

FS-Cache cannot cache data originating from any file systems. Rather, the file system driver has to have support in-built for FS-Cache. Moreover, FS-Cache won't bring in a very high performance increase on a high speed network, since the cache access speed may seem to be par with network access.

3.4.2. Different methods of back-end caching

FS-Cache is the interface between the file system and the caching mechanism. This makes the application accessing the data be agnostic to the underlying caching method.

There are two types of caching back-ends currently supported by FS-Cache, 'Cachefilesd' and 'CacheFS'. FS-Cache can use any of the caching back-end systems, as configured.

We will look into the two types of caching mechanisms available for FS-Cache.

3.4.2.1. CacheFS

CacheFS is a caching mechanism where a block device can be mounted onto the existing file system. It uses the 'mount' system call to activate the cache and make it available. It does not require any activation like 'cachefiles' (where we need to use the 'fsc' mount option), and an unmount will deactivate the cache altogether.

We won't be looking into CacheFS since it is not available in Fedora Linux for now.

3.4.2.2. Cachefiles

'Cachefiles' builds the cache on a mounted file system. This is desirable when additional disks are not easily obtainable, or a new partition cannot be created. This uses the VFS file system interfaces to get another file system such as XFS or EXT4 to do the I/O on its behalf.

'Cachefiles' is the default method supported in Fedora Linux. The following diagram Fig 3.4, shows how FS-Cache and Cachefiles co-exit between the network file system and the the back-end local file system.

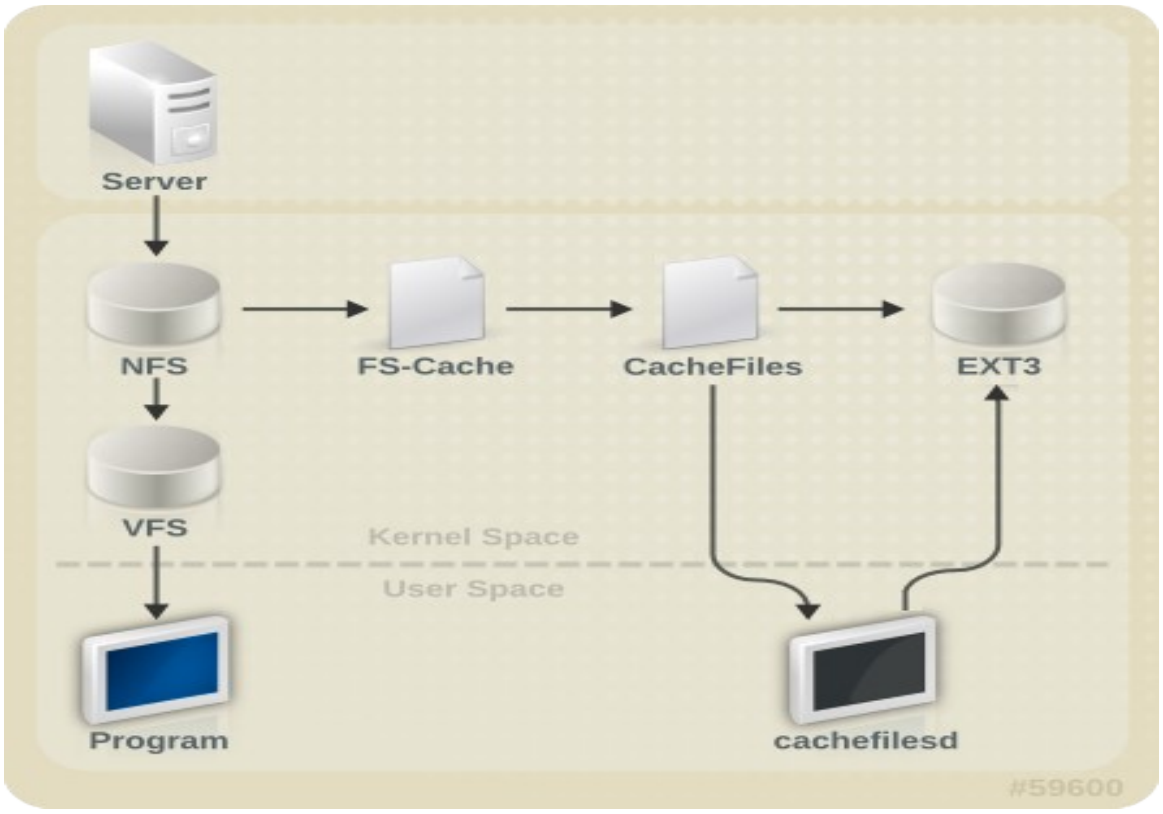

Fig 3.4 – FS-Cache and Cachefiles interaction

3.4.3. The back-end caching structure of cachefilesd

The default caching location for cachefiles is /var/cache/fscache/. Two directories named 'cache' and 'graveyard' stores the data. The cached data which is active goes into /var/cache/fscache/cache/ while the data which are set to be discarded are moved to /var/cache/fscache/graveyard. The data in 'graveyard' are usually deleted as soon as it is moved there.

3.4.3.1 Indexes and Objects

The cached data under /var/cache/fscache/cache is organized as a tree structure. There can be multiple levels/depth for a particular netfs, depending on how it's cached data is stored. The elements in each level are called Objects.

Objects are classified as two, namely 'Index object' and 'Data object'. Any member in an index other than the last index which contain the data, is called 'Index object', whereas the final member in the tree which contains cached data is called 'Data object'.

There are certain properties for objects, which we need to understand:

a) All objects other than the root object will have a parent object.

b) Any object may have as many child objects.

c) The children of an object may not always be of the same type, and there are no such restrictions.

d) Index objects may be children of other index types.

e) Non-index objects have a file size set to beyond which the pages can't be accessed, around 400Bytes.

f) Index objects may not carry data. It is the data objects that carry data.

Indexes are restricted objects as they don't contain data, and are children of other indices. Indices are used to speed up file search by splitting up the key to a file into a sequence of logical sections,

FS-Cache maintains an indexing tree in the caching back-end for all the active objects. By default, this is at /var/cache/fscache/cache/. FS-Cache creates an index object at the root of the indexing tree and it is called the 'root index'.

The children of the root index are the network file systems such as NFS, AFS, CIFS etc.. We intend to bring in FUSE in this list, with this project work. When a netfs requests caching services, an index object for that specific network file system is created under the root index.

The index created for the netfs acts as the root indice for that specific file system cache. The decision on the structure under the netfs root index is upto the netfs. Please note that this index structure is not synonymous to the actual file system hierarchy being presented to the VFS, rather the cache index would only contain objects that are actually cached. Data objects mostly form the last in the cache tree structure, and can be said as the leaves

in an upside-down tree. A data object carries several pages of data on behalf of the network file system. Holes in this are considered as pages which are not yet being fetched for that particular file. A further 'read' for that page over the network file system will fill in the holes.

3.4.4. An example showing how data from a netfs is structured, in cache

As discussed earlier in section 3.4.3.1, the cache structure of each netfs, depends on how the netfs file system decides to do it. The general idea is to maintain a tree hierarchy, but the number of indices in each level, how many levels etc.. are to be decided by the network file system.

In this section, we will try and understand how NFS has its data cache structured. We will use the command 'tree' to show a listing similar to an upside-down tree.

The command 'tree' comes in the package 'tree'. If the command is not available, install it using:

 # yum install tree -y

Once installed, you can use the command 'tree' on any folder and it will present the contents in a tree like format.

We are not looking into how to setup NFS for caching in this section. In this example, we will try to understand how the cache structure will be, for a network file system.

Imagine an NFS file system being mounted on a client machine, with caching enabled. Once the files are read from the NFS server, it gets cached locally at /var/cache/fscache/. The following gives a listing of the cache directory.

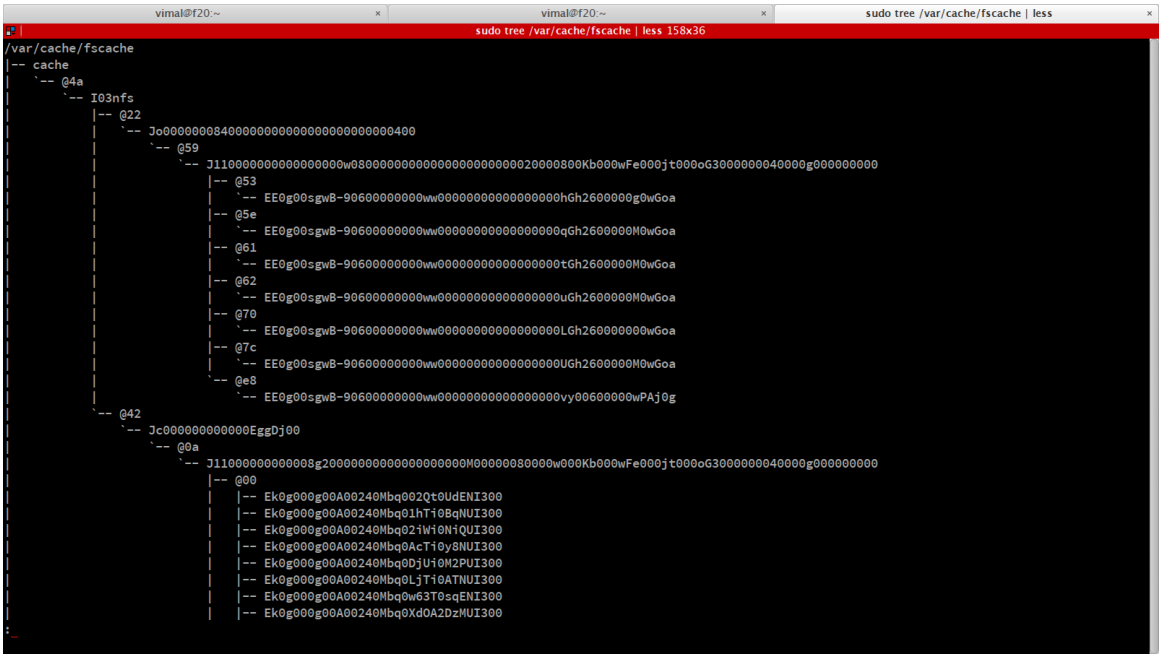

Fig 3.5 – Cachefilesd backend structure

As seen in the above example, /var/cache/fscache has a child named 'cache' which stores the actual cached objects. Under 'cache', the root index for FS-Cache can be seen named '@42'.

Under the root index, comes the main index for each network file system requesting the cache feature. For NFS, it's named as I03NFS. Under the netfs root index, indices starting with 'J' are structures for the folders in the actual file system, and further down we have the data objects with their name starting with 'E'. These contain the actual cached data.

In the next chapter we will take a detailed look into the FS-Cache API through which we can build support for caching in FUSE. We will try to understand how we can create various levels of indices as well, as well as other basic functions.

Chapter 4 - Implementation : Integrating FS-Cache in FUSE

In this chapter, we would discuss more on the idea of implementing FSCache support in FUSE, how we plan to do it using the FSCache API, important data structures in FUSE, the processes and steps followed for this integration etc..

We will be adding the code for this feature, in 'fs/fuse/fscache.c', and the needed functions or definitions would be placed in the header file 'fs/fuse/fscache.h'.

Minor additions would go into the fuse source code as well, to support this feature. The changes would be pointed out as required.

4.1. The FS-Cache netfs API

FSCache provides an API through which network file systems can use the caching facility. The API is based on three basic principles, and they are :

a) There are mainly two type of data objects, Index objects (Indices) and Data objects (files). Index objects are used by FSCache to find objects faster. The Data objects are where the cached data resides.

b) Every level of indices, be it an index object or a data object, are represented by a cookie.

c) The hierarchy of the cache is decided by the network file system. This means that, the number of indices, type etc.. under a netfs root index can be decided by the network file system.

The FSCache netfs API documentation is at [21]. The basic functions and data structures needed for the API is defined at 'linux/fscache.h', in the linux kernel source code.

4.2. Cookies

The network file system and the FSCache API interact by using cookies. The data-structure for the cookie is defined in 'include/linux/fscache.h' as following:

```
struct fscache_cookie {
        atomic_t                usage;          /* number of users of this cookie */
        atomic_t                n_children;     /* number of children of this cookie */
        atomic_t                n_active;       /* number of active users of netfs ptrs */
        spinlock_t              lock;
        spinlock_t              stores_lock;    /* lock on page store tree */
        struct hlist_head       backing_objects; /* object(s) backing this file/index */
        const struct fscache_   cookie_def *def;        /* definition */
        struct fscache_cookie   *parent;        /* parent of this entry */
        void                    *netfs_data;    /* back pointer to netfs */
```

```
    struct radix_tree_root      stores;         /* pages to be stored on this cookie */
    unsigned long               flags;
};
```

The network file system gets a cookie from FSCache when it registers itself. This cookie represents the root index of that particular network file system.

4.3. Register and Un-register with the netfs API

The network file system needs to be registered with the FSCache netfs API prior accessing any of the caching functions. In order to register the network file system, we need to pass the function 'fscache_register_netfs()', a pointer of the netfs definition.

The function 'fscache_register_netfs()' is defined, in 'include/linux/fscache.h':

```
static inline
int fscache_register_netfs(struct fscache_netfs *netfs)
{
    if (fscache_available())
        return __fscache_register_netfs(netfs);
    else
        return 0;
}
```

As said above, a pointer of the netfs definition needs to be passed to the above function, to register the netfs.

The generic data-structure for defining the netfs is defined in 'include/linux/fscache.h'.

```
struct fscache_netfs {
    uint32_t                    version;            /* indexing version */
    const char                  *name;              /* filesystem name */
    struct fscache_cookie       *primary_index;
    struct list_head            link;               /* internal link */
};
```

We use 'fscache_netfs' to define a index for the fuse file system as following, in 'fs/fuse/fscache.c'.

```
struct fscache_netfs fuse_cache_netfs = {
    .name           = "fuse",
    .version        = 0,
};
```

Once we have the root index definition as above, we pass it to the 'fscache_register_netfs' function to register the fuse netfs. The registration is done in 'fs/fuse'fscache.c' as following:

```
int fuse_register_fscache(void)
{
      return fscache_register_netfs(&fuse_cache_netfs);
}
```

Once the registration is successful, the primary index point in the netfs definition is filled with a pointer to the initial index object of the network file system. The registration should fail if there is an out of memory situation or if another network file system of the same name is registered. We are not looking into how that is done, because it is inside FSCache.

After a network file system has finished using the caching facilities, we un-register it by calling the 'fscache_unregister_netfs()' function. It is defined in 'include/linux/fscache.h', as below:

```
static inline
void fscache_unregister_netfs(struct fscache_netfs *netfs)
{
      if (fscache_available())
            __fscache_unregister_netfs(netfs);
}
```

We use this function in 'fs/fuse/fscache.c' to unregister the fuse netfs, as following:

```
/* Unregistering the netfs */
void fuse_unregister_fscache(void)
{
      fscache_unregister_netfs(&fuse_cache_netfs);
}
```

4.4. Important data structures in various components

There are a few important data structures which we need to look into, and are used frequently throughout. They mainly fall under 'fuse' and 'fscache'.

4.4.1. struct fuse_inode

'fuse_inode' is a struct defined in 'fs/fuse/fuse_i.h'. This data structure deals with the file inode which is opened through fuse.

The data structure 'fuse_inode' connects to the VFS layer in kernel through the member 'struct inode' which is a container to the 'struct inode' from include/linux/fs.h. Through this member in the struct, the inode properties such as the uid, gid, acls, access times etc.. can be accessed.

We use the struct 'fuse_inode' in the function 'fuse_fscache_enable_inode_cookie' which creates a cookie from the inode details pulled through 'struct inode'.

Due to the large number of members, we are not including the 'fuse_inode' struct in this report. It can be readily seen in the linux kernel source at 'fs/fuse/fuse_i.h'.

4.4.2. struct fuse_conn

'fuse_conn' is a data structure defined in fs/fuse/fuse_i.h' similar to 'fuse_inode'. This data-structure carries the details of a file system when it is mounted via fuse.

There are quite a few important detail that comes in this struct, such as the user id and group id of the mount, the mount options, the read and write size etc. The 'fuse_conn' struct connects to the VFS with its member 'struct super_block'.

The struct 'super_block' carries important details of the file system being mounted. Some of those are the block size, the maximum file size it supports, the inode list, the mount list, disk quota, the link count etc.

We use 'fuse_conn' in the functions 'fuse_fscache_get_subfs_cookie', 'fuse_fscache_release_subfs_cookie', 'fuse_fscache_get_mnt_cookie', 'fuse_fscache_release_mnt_cookie' etc.

Due to the large size of the struct, it is not included in this dissertation.

4.4.3. struct fscache_cookie_def

'fscache_cookie_def' is a data structure defined in 'include/linux/fscache.h'. This is the basis or acts as the blueprint for all the cookies created in the fscache layer.

This data structure defines the name of the cookie, the cookie type as in 'FSCACHE_COOKIE_TYPE_INDEX' if it is an index object, and 'FSCACHE_COOKIE_TYPE_DATAFILE' if it is a data object, an index key, file attributes of the netfs data, various indicators on the cache status etc.

We use it to create three levels of cache index, ie.. 'fuse_subfs_cache_index_def', 'fuse_mnt_cache_index_def', and 'fuse_fh_cache_index_def'.

The struct 'fscache_cookie_def' is defined as following:

```
struct fscache_cookie_def {
        char name[16];
        uint8_t type;
#define FSCACHE_COOKIE_TYPE_INDEX       0
#define FSCACHE_COOKIE_TYPE_DATAFILE    1
        struct fscache_cache_tag *(*select_cache)(
                const void *parent_netfs_data,
                const void *cookie_netfs_data);
        uint16_t (*get_key)(const void *cookie_netfs_data,
```

```
                void *buffer,
                uint16_t bufmax);
    void (*get_attr)(const void *cookie_netfs_data, uint64_t *size);
    uint16_t (*get_aux)(const void *cookie_netfs_data,
                void *buffer,
                uint16_t bufmax);
    enum fscache_checkaux (*check_aux)(void *cookie_netfs_data,
                        const void *data,
                        uint16_t datalen);
    void (*get_context)(void *cookie_netfs_data, void *context);
    void (*put_context)(void *cookie_netfs_data, void *context);
    void (*mark_page_cached)(void *cookie_netfs_data,
                    struct address_space *mapping,
                    struct page *page);
    void (*now_uncached)(void *cookie_netfs_data);
};
```

4.5. Create the indices

We create four level of indices in this implementation. All the three indices other than the initial index is based on the struct 'fscache_cookie_def' defined in 'include/linux/fscache.h'.

4.5.1. First level index – 'struct fuse_cache_netfs'

This is the first level index, and is a copy of the 'fscache_netfs' (include/linux/fscache.h). It is the index that is used for the netfs registration, mentioned in section 4.3.

```
struct fscache_netfs fuse_cache_netfs = {
    .name           = "fuse",
    .version        = 0,
};
```

This struct contains the netfs name and a version number. The string 'name' is used to create the initial index under /var/cache/fscache/cache/.

4.5.2. Second level index – 'struct fuse_subfs_cache_index_def'

The second index denotes the sub – file system that comes under fuse. Here we use 'glusterfs' since we are interested in caching contents from a glusterFS volume mounted locally over fuse. We base the second level index on the struct 'fscache_cookie_def'. This data structure is defined in 'include/linux/fscache.h' and we detail it in section 4.2.2.1.

```
struct fscache_cookie_def fuse_subfs_cache_index_def = {
    .name       = "fuse.subfs",
    .type       = FSCACHE_COOKIE_TYPE_INDEX,
    .get_key    = fuse_fs_cache_get_key,
    .get_aux    = fuse_fs_cache_get_aux,
```

```
    .check_aux      = fuse_fs_cache_check_aux,
};
```

The name is set to 'fuse.subfs' which will get translated to 'fuse.glusterfs'. The type is defined as 'FSCACHE_COOKIE_TYPE_INDEX' since this is an index cookie and not a data cookie.

The function 'fuse_fs_cache_get_key' is used to set the key for the index and later is filled with the cache content. The key can be seen as a table where the cache is filled by another function, which we will be looking into later.

We are not interested in the get_aux and check_aux functions right now, and has set aside this for a future implementation. These are not important to get a working prototype.

4.5.3. Third level index – 'struct fuse_mnt_cache_index_def'

The third level index named 'fuse_mnt_cache_index_def' is based on the struct 'fscache_cookie_def'.

This index creates an entry to cache details of the fuse mount point, on the local machine. For NFS this can be a share name, but for glusterfs this is a volume id, preferably called a UUID. Each glusterfs volume will have a UUID that distinguish it from another volume.

```
struct fscache_cookie_def fuse_mnt_cache_index_def = {
    .name           = "fuse.mnt",
    .type               = FSCACHE_COOKIE_TYPE_INDEX,
    .get_key        = fuse_mnt_cache_get_key,
    .get_aux        = fuse_mnt_cache_get_aux,
    .check_aux       = fuse_mnt_cache_check_aux,
};
```

The type is set to FSCACHE_COOKIE_TYPE_INDEX since this is an index object and not a data object. The main function that creates the key for to store the cache is 'fuse_mnt_cache_get_key'. We are not interested in the aux functions for now, since this is not mandatory to get a working prototype.

4.5.4. Fourth level index – 'struct fuse_fh_cache_index_def'

The fourth and final index level is named 'fuse_fh_cache_index_def' is based on the struct 'fscache_cookie_def' and this creates an index which carries details of the actual data, in a file.

Each file in a GlusterFS volume contains a GFID which is a 128 bit identifier for the file. This is synonymous to the file inode number. For this prototype, we are using an inode number which is readily available from the 'struct fuse_inode' defined in 'fs/fuse/fuse_i.h'. Moving solely to a GFID is left for a future implementation. The struct for the final index

is defined as:

```
struct fscache_cookie_def fuse_fh_cache_index_def = {
      .name                 = "fuse.fh",
      .type                 = FSCACHE_COOKIE_TYPE_DATAFILE,
      .get_key          = fuse_fh_cache_get_key,
      .get_attr             = fuse_fh_cache_get_attr,
      .get_aux          = fuse_fh_cache_get_aux,
      .check_aux            = fuse_fh_cache_check_aux,
      .now_uncached     = fuse_fh_cache_now_uncached,
};
```

The main difference compared to other indices is the type. This is set to 'FSCACHE_COOKIE_TYPE_DATAFILE'. This denotes that this index is a data object rather than an index object like the previous indices. We will look into the functions 'fuse_fh_cache_get_key' which creates the index key and 'fuse_fh_cache_get_attr' which provides the file attributes. As in the indices before, we are not interested in the aux functions since they are not needed for this prototype.

4.6. Functions defined in each indices

In this section, we will try to understand the functions defined in each index level. These are some of the important building blocks in this prototype.

4.6.1. The first index – 'fuse_cache_netfs()'

There are no specific functions which are needed in the initial level. This is level is not considered as an index object or a data object, and is not based on the 'struct fscache_cookie_def'. Hence this index does not have a type or other members of the 'fscache_cookie_def'.

This index simply contains the name of the netfs and a version number.

4.6.2. The second index - 'fuse_subfs_cache_index_def()'

The second index is based on the struct 'fscache_cookie_def' (include/linux/fscache.h). We are interested in a single function 'fuse_fs_cache_get_key' for this level.

This function fills a void pointer named 'buffer' with the file system subtype. 'buffer' should end up being filled up with the string 'fuse.glusterfs'. The function is defined as:

```
/* a) Set the key for the index entry */
static uint16_t fuse_fs_cache_get_key(const void *cookie_netfs_data,
                                      void *buffer, uint16_t bufmax)
{
```

```
/* This will create the 2nd level index, will fill 'buffer'
* with the string 'fuse.glusterfs', and return its length
*/
        const struct fuse_conn *fc = cookie_netfs_data;
        struct fuse_fscache_subfs_key *key = buffer;

        /* "klen" will contain the length of the key */
        uint16_t klen;
        klen = sizeof(struct fuse_fscache_subfs_key);

        if (klen > bufmax)
                return 0;

        /* Copy the contents of 'fc->sb->s_subtype' to 'buffer'
         * 'buffer' should contain 'fuse.glusterfs'.
         */
        strncpy(key->subtype, fc->sb->s_subtype, sizeof(key->subtype));
        return klen;
}
```

As seen above, the function 'fuse_fs_cache_get_key' takes three arguments, two void pointers 'cookie_netfs_data' and 'buffer', and an unsigned 16-bit integer 'bufmax'.

The main work done is by the 'strncpy' function which copies 'fc->sb->s_subtype' to 'key->subtype'.

'fc' is a pointer to the struct 'fuse_conn' (fs/fuse/fuse_i.h), and 'sb' is a pointer to the struct 'super_block' (include/linux/fs.h) inside 'fuse_conn'. The 'super_block' struct has a member named 's_subtype' which is a string containing the name of the file system.

The function returns the length of the string 'fuse.glusterfs' and also fills the buffer pointer with the string.

4.6.3. The third index – 'fuse_mnt_cache_index_def()'

The third index 'fuse_mnt_cache_get_key' works similar to the second index, by copying a string to a void pointer and returning its length.

This index deals with the glusterfs volume-id, also called UUID. Currently there is a drawback in getting a volume-id directly from fuse and hence we use a static volume id defined in this function.

Since a volume-id remains static for a particular volume, we would be using a fixed UUID in this function which comes from a volume already up and running on the two gluster virtual machines.

```
static uint16_t fuse_mnt_cache_get_key(const void *cookie_netfs_data,
                                       void *buffer, uint16_t bufmax)
{
/* We'll use a fixed volume-id for now, since we don't have a
 * singular way to get it from fuse
 */
      const char* uuid = "e545e1ea-e932-45ad-a003-2e91bacea403";
      struct fuse_fscache_mnt_key *key = buffer;
      uint16_t klen = sizeof(struct fuse_fscache_mnt_key);

      if (klen > bufmax)
            return 0;
      strncpy(key->fsid, uuid, strlen(uuid));
      return klen;
}
```

The volume-id we use would remain static for this prototype, as shown above. Changing a volume-id would require the fuse module be rebuilt, for now. Moving this to take in the volume-id of any volumes that are being mounted, is set for a future, more robust implementation.

'strncpy' is the important part in this function where the 'uuid' is copied to the 'fsid' member of the struct 'key' (struct fuse_fscache_mnt_key) and it returns the size of klen, ie.. the size of the struct fuse_fscache_mnt_key.

4.6.4. The fourth index – 'fuse_fh_cache_index_def()'

The fourth index 'fuse_fh_cache_index_def' returns the length of the file inode. It is created similar to the second and third index, from 'fscache_cookie_def'

```
static uint16_t fuse_fh_cache_get_key(const void *cookie_netfs_data,
                                      void *buffer, uint16_t bufmax)
{
      const struct fuse_inode *fi = cookie_netfs_data;
      struct fuse_fscache_fh_key *key = buffer;
      uint16_t fhlen = sizeof(struct fuse_fscache_fh_key);
      if (fhlen > bufmax)
            return 0;
      key->ino = fi->orig_ino;
      return fhlen;
}
```

This function creates a pointer to the struct 'fuse_inode', named 'fi', as well as another pointer named 'key' to the struct 'fuse_fscache_fh_key'. The member 'orig_ino' which is the inode number in the 'fuse_inode' struct is then copied to the 'ino' member of the struct 'key'. After this, the 16bit unsigned int 'fhlen' is returned which contains the size of the struct 'fuse_fscache_fh_key'.

4.7. Create cookies for each index levels.

These are three functions, one for each level of the index, starting from the second index. This creates the cookies using the 'fscache_acquire_cookie()' function, which is defined in include/linux/fscache.h.

```
/* a) Construct a cookies for the initial index, ie.. glusterfs FS level */
void fuse_fscache_get_subfs_cookie(struct fuse_conn *fc)
{
	fc->fscache.subfs = fscache_acquire_cookie(
				fuse_cache_netfs.primary_index,
				&fuse_subfs_cache_index_def, fc, true);
}

/* b) Construct a cookie for the second index level, ie.. the mount point or vol-id*/
void fuse_fscache_get_mnt_cookie(struct fuse_conn *fc)
{
	fc->fscache.mnt = fscache_acquire_cookie(fc->fscache.subfs,
				&fuse_mnt_cache_index_def, fc, true);
}

/* c) Construct a cookie for the third index level, ie.. the inode/fsid */
static void fuse_fscache_enable_inode_cookie(struct inode *inode)
{
	struct fuse_inode *fi = get_fuse_inode(inode);
	struct fuse_conn *fc = get_fuse_conn_super(inode->i_sb);

	if (fi->fscache)
		return;

	if (fc->flags & FUSE_MOUNT_FSCACHE) {
		fi->fscache = fscache_acquire_cookie(fc->fscache.mnt,
				&fuse_fh_cache_index_def, fi, true);
	}
}
```

4.8. Fill the indexes with the cache

This is one of the most important functions in this feature. This function writes the data from the netfs and fills the cache indices with it.

The function 'fuse_readpage_to_fscache()' defined in 'fs/fuse/fscache.h' inturn calls '__fuse_readpage_to_fscache()' in 'fs/fuse/fscache.c'. It reads the memory pages into cache. The main data structures called here are 'struct inode', 'struct page', and 'struct fuse_inode'.

The data is read from the netfs as pages, ie.. chunks of fixed size. The pages are pulled from the struct 'struct page' and is written to the cache by the function 'fscache_write_page()' (include/linux/fscache.h). The reader function is defined as:

```
void __fuse_readpage_to_fscache(struct inode *inode, struct page *page)
{
      struct fuse_inode *fi = get_fuse_inode(inode);
      int ret;

      ret = fscache_write_page(fi->fscache, page, GFP_KERNEL);
      printk(KERN_WARNING "fuse_readpage_to_fscache: fscache=%p, page=%p, gfp=%u\n", fi->fscache, page, GFP_KERNEL);

      if (ret !=0)
            fscache_uncache_page(fi->fscache, page);
}
```

4.9. Return cached data from the local storage.

We saw the function that fetches the data pages and copies them to the cache. The next important function is the one that fetches the data from the cache at the time of reads.

The function 'fuse_readpages_from_fscache()' is defined in 'fs/fuse/fscache.h' and in turn calls '__fuse_readpages_from_fscache()' from 'fs/fuse/fscache.c'.

This function checks for the presence of the member 'fi->fscache' which should be filled by 'fuse_readpages_to_fscache()' discussed in section 4.8. If it has been filled, it calls 'fuse_readpages_from_fscache()' from 'fs/fuse/fscache.c'.

```
* Retrieve a set of pages from FS-Cache
*/
int __fuse_readpages_from_fscache(struct inode *inode,
                        struct address_space *mapping,
                        struct list_head *pages,
                        unsigned *nr_pages)
{
      int ret;
      struct fuse_inode *fi = get_fuse_inode(inode);

      printk(KERN_WARNING "%s: (0x%p/%u/0x%p)\n", __func__,
          fi->fscache, *nr_pages, inode);

      ret = fscache_read_or_alloc_pages(fi->fscache, mapping,
                              pages, nr_pages,
                              fuse_readpage_from_fscache_complete,
                              NULL,
                              mapping_gfp_mask(mapping));
      switch (ret) {
      case 0: /* read submitted to the cache for all pages */
            printk(KERN_WARNING "%s: submitted\n", __func__);
            return ret;

      case -ENOBUFS:  /* some pages are not cached and can't be */
      case -ENODATA:  /* some pages are not cached */
           printk(KERN_WARNING "%s: no page\n", __func__);
```

```
            return 1;
        default:
             printk(KERN_WARNING "unknown error ret = %d\n", ret);
        }
         return ret;
}
```

4.10. The code for the working prototype

Here we list the code that went into the project, as git one-liner patches. Unfortunately, the whole code cannot be included in this dissertation due to the page limits we have.

4.10.1. A git one-liner list of the patches

Git can print the patch number as well as the patch subject in a single line, and it serves to present a short detail about the patches. We will list all the patches that has gone into the code.

```
# git log --oneline master..HEAD
84eb0de fuse_fh_cache_get_attr() gets a fuse_inode as parameter
22d012e fill the cache when calling fuse_send_write_pages()
a4b5a50 use the fuse_inode->nodeid instead of orig_ino
6731880 Initialize the subfs and mnt indexes
ef099b4 Add/rework fuse_read_to_fscache()
02db47e initialize the fuse_inode->fscache pointer to NULL
0ad9e45 move DEBUG messages to WARNING for testing
ab24c05 add fuse_fscache_reset_inode_cookie()
545f4f4 add fuse_readpages_from_fscache()
2b5b665 Added the fuse_fscache_disable_inode_cookie() function
4c47e6b Slight changes in function declarations
1311194 Added the fuse_fscache_set_inode_cookie() function
d4578a6 Comments in fs/fuse/fscache.c :
07c8b76 Added the function fuse_fscache_set_inode_cookie(), and comments in fs/fuse/file.c
3ef7c24 Small changes in one commit
3d93400 Fill the keys/buffers with a structure, not a string/integer
01e2525 Add fuse_readpage_to_fscache()
b58c10b cleanup fscache.h
de5e84e9 (un)register fscache when an error occurs
829f1db use memcpy() for filling the buffer with the inode in fuse_fh_cache_get_key()
20aa09f move fuse_readpage_to_fscache() to fuse_do_readpage()
b2e2947 Moved the function fuse_readpage_to_cache() to fscache.c, and left the declaration in
fscach.h
2590a7a Added fuse_readpage_to_fscache() in fscache.c
22cb781 Comments for fi->orig_ino
768b704 Set 'fc.fscache->subfs' to 'fc->fscache.subfs' in 'fuse_fscache_get_subfs_cookie()', and set
'buffer = (void*) fi->orig_ino' in 'fuse_fh_cache_get_key()'
347bb1d Changed 'fc->fscache.subfs' to 'fc.fscache->subfs' in fscache.c
957a5bf Cleaned the declarations from the previous function bodies, removed 'inline' keyword from
the declarations
79970cd Removed the register/unregister functions from fscache.h, maintained it as declarations
```

2272f2a Added a printk to log a debug message in 'dmesg' output, to understand the opts, token etc..
75b6290 Added fuse_register_fscache() in fuse_init(), to register it at startup
5e6dcb2 Comments and cleanup
32c8ee7 Functions for index levels
2279097 Ordering of the functions, in each index level
c8b6923 fuse/fscache: add some get_*_cookie() functions
faf5846 fuse/fscache: add "fsc" mount option
e5061d0 fuse: initial skeleton for FS-Cache support

4.10.2. An overview of the total code changes

The following git output lists the files, and the total changes that went into the files in the course of this project work.

```
# COLUMNS=80 git diff --stat master..HEAD
fs/fuse/Kconfig   |   9 +
fs/fuse/Makefile  |   1 +
fs/fuse/file.c    |  35 +++-
fs/fuse/fscache.c | 501 ++++++++++++++++++++++++++++++++++++++++++++++++++++++
fs/fuse/fscache.h | 111 ++++++++++++
fs/fuse/fuse_i.h  |  17 ++
fs/fuse/inode.c   |  43 ++++-
7 files changed, 709 insertions(+), 8 deletions(-)
```

The above statistics show that 7 files have been changed in the course of this project work, 709 lines of code has been added, and 8 lines of code has been deleted. It also shows the file name as well as the number of lines that has been added in each file

Chapter 5 - Testing

5.1. Scenario

In this chapter, we will install the test environment which comprises of the two GlusterFS servers and the client machine, compile the fuse source code on the client machine, mount the glusterfs volumes, and test the caching.

5.2. Prepare the Gluster cluster nodes

This section talks about how to install and configure glusterfs nodes for testing.

5.2.1. Install the GlusterFS nodes.

As mentioned in section 2.2, we will install two Fedora 20 virtual machines using virt-manager[9] from the Fedora ISO[13]. The Fedora installation guide is referred at [14].

Both the machines will have 10GB of hard disk space, and 1GB of memory. Additionally, each virtual machine will have a secondary hard disk of 1GB, which will act as a brick to comprise the cluster volume. The following diagrams walk through the process of installation

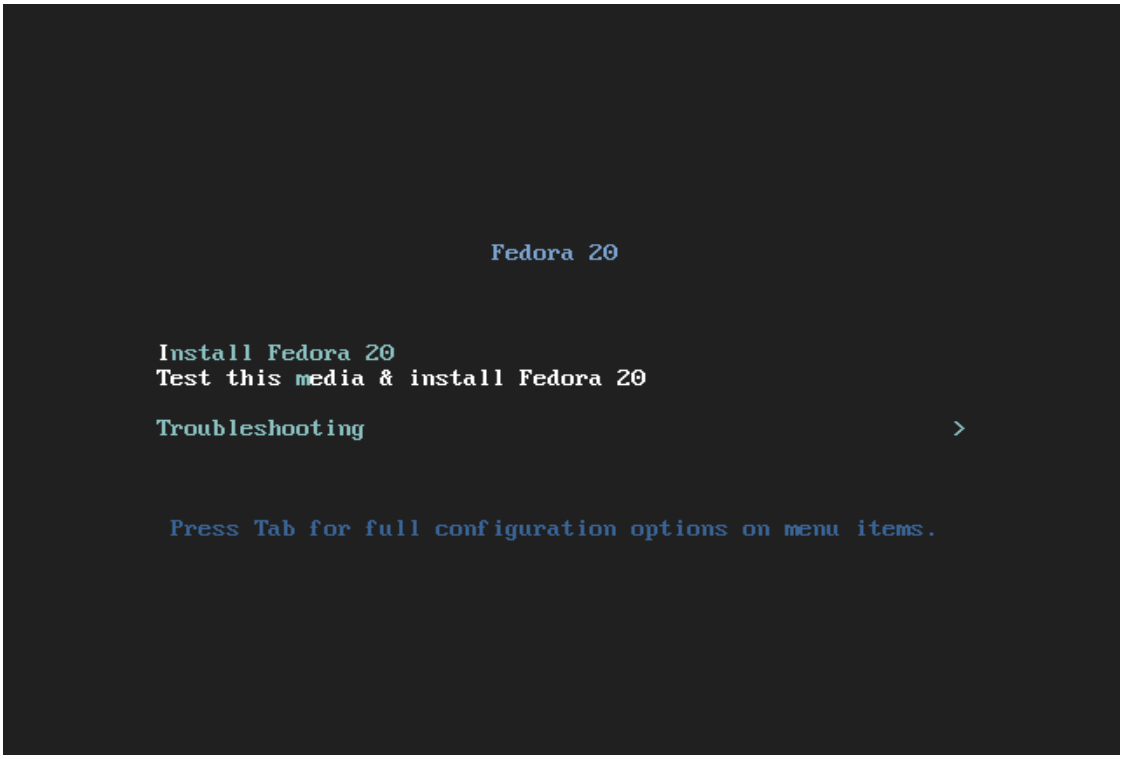

Fig 5.1. The Fedora 20 boot screen.

The screen in the previous page shows the boot screen. Once the virtual machine is booted from the Fedora 20 ISO, the user will be greeted with the boot screen, from where they have the choice of either install the OS onto the hard disk, or test this by running a live image in memory.

We follow the steps in the Fedora installation guide [15] to go through the process and finish the installation.

Once finished, the machine reboots and we are greeted with the login screen shown below:

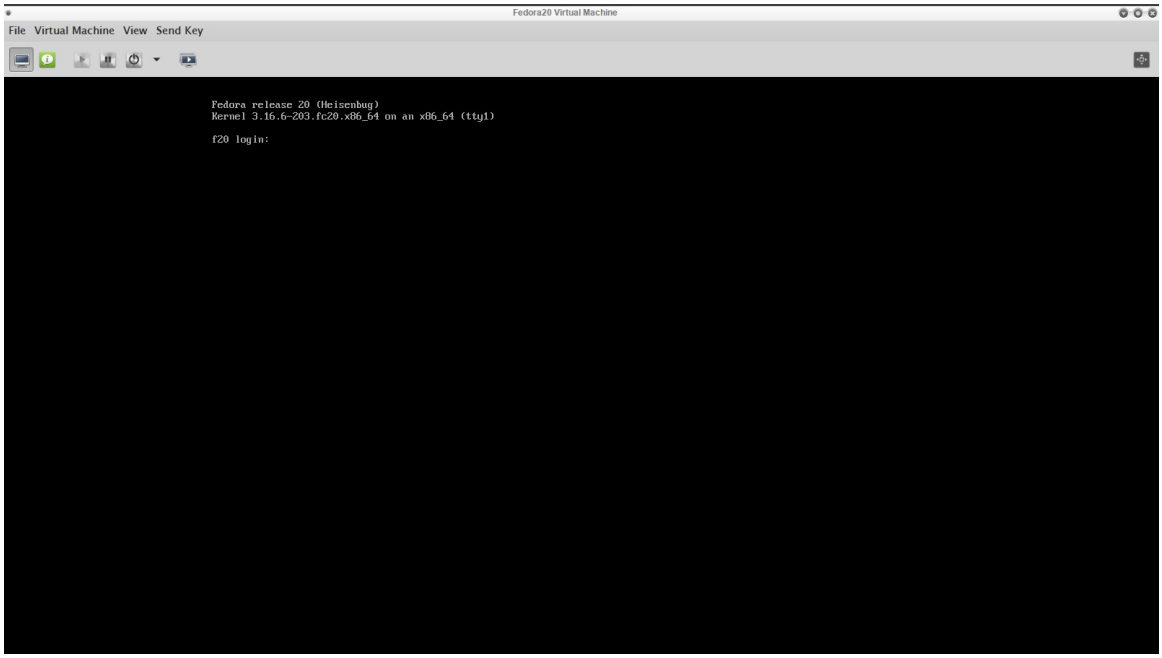

Fig 5.2. The login prompt

5.2.2. Create the back-end bricks

After the Fedora virtual machines are installed, we go ahead and add a single 1GB disk to each machine. We will be using virt-manager to add the disk. The following diagrams show the process of adding a extra hard disk.

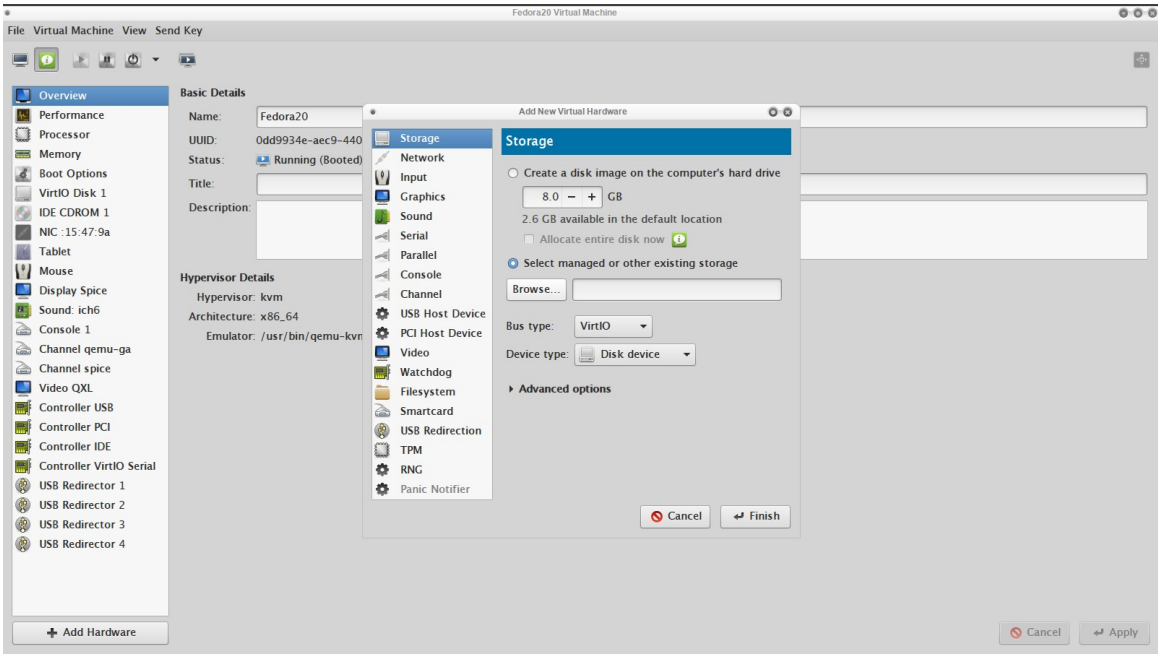

Fig 5.3. Adding a virtual hard-disk to the existing Fedora installation.

The installation will be on /dev/vda and the new disk will be detected as /dev/vdb.

The new disk will need to be partitioned and formatted with XFS file system. The steps can be summarized as following:

a) Partition the disk /dev/vdb using 'fdisk'.

 # fdisk /dev/vdb

Once partitioned, the partition will be named as /dev/vdb1. The next step would be to create an XFS file system on top of /dev/vdb1, with an inode size of 512 Bytes.

 # mkfs.xfs -i size=512 /dev/vdb1

5.2.3. Install the glusterfs server packages

In order to create a glusterfs server, we will need to install the glusterfs-server packages and its dependencies. Install the same using yum:

 # yum install glusterfs-server -y

After the glusterfs-server package and its dependencies are installed, start the glusterd service.

5.2.4. The glusterd process

The 'glusterd' process when started runs in the background as a daemon process. This is the main process which enables the glusterfs cluster and it enables dynamic configuration changes as well as monitor the nodes.

We need to start the glusterd daemon in order to configure the nodes as part of the cluster.

 # service glusterd start

Once the 'glusterd' service is started, confirm if it is running using the commands

 # service glusterd status

5.2.5. Create a trusted pool, or peer group.

The next step would be to create a trusted storage pool from which the cluster is to be formed. A trusted storage pool is a network of storage servers. When the first server starts, the trusted pool consists of just that machine. Other machines are added one by one as needed. The following command adds a gluster node to the trusted pool.

 # gluster peer probe <IP Address>

The 'gluster peer probe' command should output 'Probe successful', if the peer was added properly. Once a node is successfully added, the peer list can be printed out using :

 # gluster peer status

This should print the other nodes in the peer group, other than the localhost. The output should look similar to:

Number of Peers: 1

Hostname: server2
Uuid: 5e987bda-16dd-43c2-835b-08b7d55e94e5
State: Peer in Cluster (Connected)

The peer status will print the number of peers, the hostname, the UUID of the server, and the status of the peer in the group.

5.2.6. Create a volume

A glusterfs volume is a collection of bricks, where a brick is an XFS file system mounted onto the main root file system. We will be using the additional 1GB disk to act as a brick.

To create the volume, we will use two bricks existing on 'server'1' and 'server2'. The

bricks are mounted at /bricks/vol1 and /bricks/vol2 on server1 and server2 respectively.

```
# gluster volume create vol1 server1:/bricks/vol1 server2:/bricks/vol2
```

Once the volume is created, the status can be tested as following:

```
# gluster volume info
Volume Name: vol1
Type: Distribute
Status: Created
Number of Bricks: 2
Transport-type: tcp
Bricks:
Brick1: server1:/exp1/brick
Brick2: server2:/exp2/brick
```

In order to mount and access the volume on a client, the volume needs to be started. We will see how to start the volume in the next section.

5.2.7. Start the volume

A volume needs to be started after creation, in order to be accessed from the client machine. In order to start a volume, use:

```
# gluster volume start <volume-name>
```

This should start the volume, and 'gluster volume status' should show it as 'Started'.

```
# gluster volume start vol1
Starting vol1 has been successful
```

A 'gluster volume info' should show the status as 'Started'.

```
# gluster volume info
Volume Name: vol1
Type: Distribute
Status: Started
Number of Bricks: 2
Transport-type: tcp
Bricks:
Brick1: server1:/exp1/brick
Brick2: server2:/exp2/brick
```

5.3. Prepare the client machine

The next step in the process is to prepare the client machine. We will install a third virtual machine with Fedora 20, similar to the glusterfs nodes. The client machine has the same configuration as the server nodes, ie.. 10GB of disk space and 1GB of memory.

We will not discuss the installation process since it is the same as the GlusterFS nodes discussed in section 5.2.1.

5.3.1. Install the glusterfs client packages

The two main differences between the GlusterFS server nodes and the client are:

a) The client node does not need an extra 1GB hard disk to act as a brick, since this is not part of a cluster.

b) The glusterfs-server packages are not needed. Instead we will install the glusterfs-fuse and glusterfs packages.

```
# yum install glusterfs-fuse glusterfs -y
```

This should install the client side packages, as well as the dependencies.

5.4. Build and load the 'fuse' kernel module

The next step would be to build and load the custom fuse.ko module on the client machine. The code that goes to build the support for fscache integration is under fs/fuse/ and hence a module re-build is needed to get the new features.

To build the module for the current booted kernel:

a) Change directory to the kernel source

```
# cd /Kernel-source/kernel-3.16.fc20/linux-3.16.6-200.fc20.x86_64
```

b) Run a 'make clean' to clean off the earlier object files (*.o) created for each C program files. The object files are created for each C program files at the time of complilation, and linked to the final .ko module file.

NOTE: The 'Makefile' at the location specified at the location with '-C' will have a definition on what to do if a 'make clean' is run.

```
# make -C /lib/modules/$(uname -r)/build M=$PWD/fs/fuse/ clean
```

c) Compile the source code in /fs/fuse/ as following:

```
# make CONFIG_FUSE_FSCACHE=y EXTRA_CFLAGS=-DCONFIG_FUSE_FSCACHE -C
/lib/modules/$(uname -r)/build M=$PWD/fs/fuse/
```

'CONFIG_FUSE_FSCACHE=y' is the directive added in '.config'. We pass it manually so that it is forcefully taken into consideration during the build.

The compile process should finish without any errors. Warning messages about unused struct members are fine though.

d) Once the compilation works fine, inspect the module. This should fuse.ko' has a dependency on 'fscache', so load it initially.

```
# modinfo fs/fuse/fuse.ko
```

The 'modinfo' command should list the details of the module as:

```
filename:       /Kernel-source/kernel-3.16.fc20/linux-3.16.6-200.fc20.x86_64/fs/fuse/fuse.ko
alias:          devname:fuse
alias:          char-major-10-229
alias:          fs-fuseblk
alias:          fs-fuse
license:        GPL
description:    Filesystem in Userspace
author:         Miklos Szeredi <miklos@szeredi.hu>
alias:          fs-fusectl
depends:        fscache
vermagic:       3.16.6-200.fc20.x86_64 SMP mod_unload
parm:           max_user_bgreq:Global limit for the maximum number of backgrounded requests an
unprivileged user can set (uint)
parm:           max_user_congthresh:Global limit for the maximum congestion threshold an
unprivileged user can set (uint)
```

e) Load both the 'fscache' and 'fuse' modules

```
# modprobe fscache
# insmod fs/fuse/fuse.ko
```

GlusterFS needs the fuse module by default, since it works over fuse. We would also need the 'fscache' module to be loaded since this prototype needs fscache to cache the contents in the caching backend.

Since 'fuse' depends on 'fscache', load 'fscache' first and then 'fuse'.

5.5. Configure FS-Cache to use cachefilesd

As discussed in section 3.4, FSCache is an interface between the network file system and the actual caching backend, ie.. Cachefilesd or CacheFS. We will be using Cachefilesd to work as a caching backend.

Prior using Cachefilesd, it needs to be configured to get the caching back-end working as desired. Install the package using yum, to start with.
a) Install 'cachefilesd' package using 'yum':

```
# yum install cachefilesd -y
```

The package installation will create the file /etc/cachefilesd.conf which acts as the configuration file for the back-end.

b) Open /etc/cachefilesd.conf and set a proper path for the directive 'dir'. This would be the location where Cachefilesd will cache the data. The default value is set to '/var/cache/fscache/', which we will continue to use.

 # vi /etc/cachefilesd.conf

Set 'dir /var/cache/fscache/'.

c) Once the path is set and the configuration file closed, restart the 'cachefilesd' service, and check the status.

 # sudo systemctl start cachefilesd

 # sudo systemctl status cachefilesd

This should be enough for 'cachefilesd' to be configured as a caching back-end. If the service 'cachefilesd' has started successfully, the folders /var/cache/fscache/cache and /var/cache/fscache/graveyard should be created.

The cached data would be present in /var/cache/fscache/cache and any cache that are for purging will be moved to /var/cache/fscache/graveyard.

5.6. Mount the glusterfs volume using the glusterfs native client

We will use the GlusterFS native client to mount the GlusterFS volume from the cluster, on the client. This is a user-space client running in user-space. This section will look into how the native client can be used to mount and access the GlusterFS volumes, how to verify if it is mounted, and how to access the volume.

The following 'mount' command will mount the GlusterFS volume on the client machine.

 # mount -t glusterfs -o log-file=/var/log/gluster.log server1:/vol1 /mnt/glusterfs

The above command will mount the volume 'vol1' which is available via the two GlusterFS servers 'server1' and 'server2', at /mnt/glusterfs.

The log file is set to /var/log/gluster.log and is not a mandatory mount option. By default, the logs would go inside /var/log/glusterfs/mnt_glusterfs.log.

Once the volume is mounted, the volume can be read and written to from /mnt/glusterfs.

5.7. Write/Read data to the glusterfs mount point

We will read/write to the GlusteFS mount point (/mnt/glusterfs) and check if the contents are cached in /var/cache/fscache/cache/.

In this prototype, we have only looked into how a read cache can be implemented, and not on write caching. This means that data which are written to the glusterfs mount point is not cached. Only reads from the mount point are cached.

a) Prior reading any data via the glusterfs mount point, check the FSCache stats in /proc/fs/fscache/. All the indexes should be empty, as shown.

```
# cat /proc/fs/fscache/stats
FS-Cache statistics
Cookies: idx=0 dat=0 spc=0
Objects: alc=0 nal=0 avl=0 ded=0
ChkAux : non=0 ok=0 upd=0 obs=0
Pages  : mrk=0 unc=0
Acquire: n=0 nul=0 noc=0 ok=0 nbf=0 oom=0
Lookups: n=0 neg=0 pos=0 crt=0 tmo=0
Invals : n=0 run=0
Updates: n=0 nul=0 run=0
Relinqs: n=0 nul=0 wcr=0 rtr=0
AttrChg: n=0 ok=0 nbf=0 oom=0 run=0
Allocs : n=0 ok=0 wt=0 nbf=0 int=0
Allocs : ops=0 owt=0 abt=0
Retrvls: n=0 ok=0 wt=0 nod=0 nbf=0 int=0 oom=0
Retrvls: ops=0 owt=0 abt=0
Stores : n=0 ok=0 agn=0 nbf=0 oom=0
Stores : ops=0 run=0 pgs=0 rxd=0 olm=0
VmScan : nos=0 gon=0 bsy=0 can=0 wt=0
Ops    : pend=0 run=0 enq=0 can=0 rej=0
Ops    : dfr=0 rel=0 gc=0
CacheOp: alo=0 luo=0 luc=0 gro=0
CacheOp: inv=0 upo=0 dro=0 pto=0 atc=0 syn=0
CacheOp: rap=0 ras=0 alp=0 als=0 wrp=0 ucp=0 dsp=0
```

We confirm that all the values are zero in the above output, which means that the caching has not yet happened. Test the caching as shown in the next page.

a) Start cachfilesd service

```
# service cachefilesd start
```

b) Mount the glusterfs volume with the 'fsc' mount option

```
# mount -t glusterfs 192.168.122.84:/vol1 /fscache-test -o log-level=TRACE,fsc
```

c) Write a few text files as well as data files onto the mount point.

```
# for i in `seq 1 10`; do echo "Testing fscache-fuse integration.: file$i" > /mnt/glusterfs/file$i ; done
# dd if=/dev/zero of=/mnt/glusterfs/imagefile.img bs=1M count=1
```

d) Read the files.

```
# cat /mnt/glusterfs/file* > /dev/null
# cat /mnt/glusterfs/imagefile.img > /dev/null
```

Reading the files should populate the FSCache statistics and object counts.

```
# cat /proc/fs/fscache/stats
FS-Cache statistics
Cookies: idx=2 dat=13 spc=0
Objects: alc=16 nal=0 avl=16 ded=6
ChkAux : non=0 ok=2 upd=0 obs=0
Pages  : mrk=12 unc=6
Acquire: n=15 nul=0 noc=0 ok=15 nbf=0 oom=0
Lookups: n=16 neg=14 pos=2 crt=14 tmo=0
Invals : n=0 run=0
Updates: n=0 nul=0 run=0
Relinqs: n=6 nul=0 wcr=0 rtr=6
AttrChg: n=0 ok=0 nbf=0 oom=0 run=0
Allocs : n=0 ok=0 wt=0 nbf=0 int=0
Allocs : ops=0 owt=0 abt=0
Retrvls: n=12 ok=0 wt=5 nod=12 nbf=0 int=0 oom=0
Retrvls: ops=12 owt=1 abt=0
Stores : n=12 ok=12 agn=0 nbf=0 oom=0
Stores : ops=12 run=24 pgs=12 rxd=12 olm=0
VmScan : nos=0 gon=0 bsy=0 can=0 wt=0
Ops    : pend=1 run=24 enq=24 can=0 rej=0
Ops    : dfr=0 rel=24 gc=0
CacheOp: alo=0 luo=0 luc=0 gro=0
CacheOp: inv=0 upo=0 dro=0 pto=0 atc=0 syn=0
CacheOp: rap=0 ras=0 alp=0 als=0 wrp=0 ucp=0 dsp=0
```

The figure below shows the output of /proc/fs/fscache/objects. This details the name of the index keys, and the data each index contains.

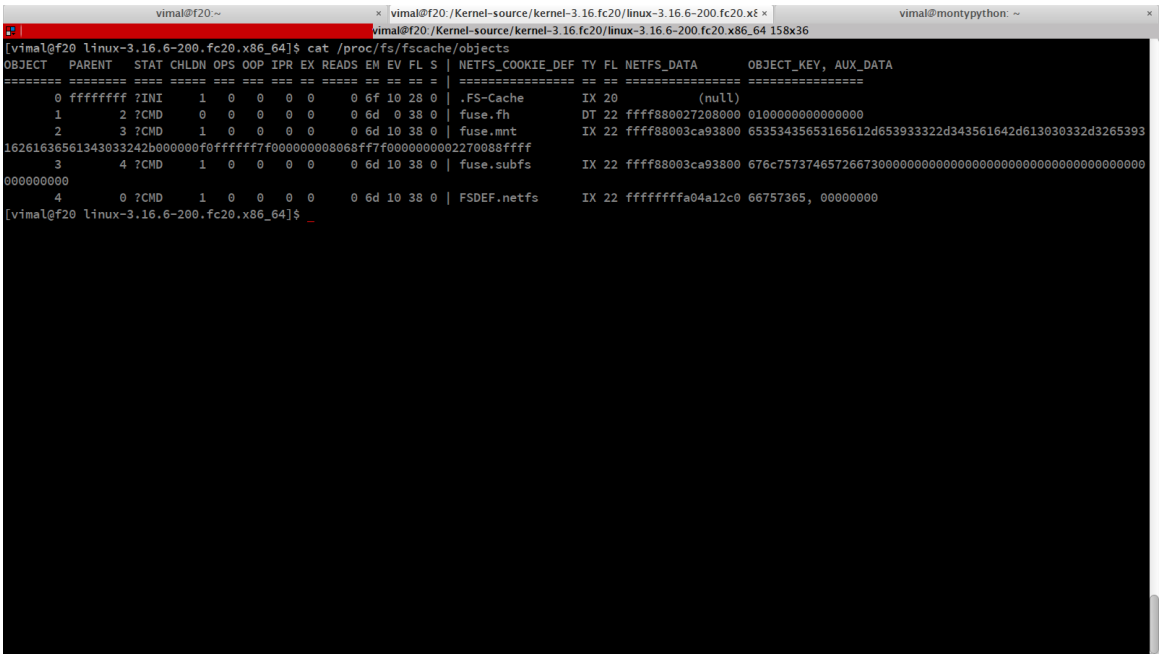

Fig 5.4. /proc/fs/fscache/objects

The above statistics show that FSCache has indeed populated the stats as well as the object details.

In order to confirm if the keys are indeed from the actual glusterfs mount, we use the following python program to decode the keys.

```
#decode-key.py
fuse_mnt ='<value>'
fuse_subfs = '<value>'
FSDEF_netfs = '<value>'

def decode_hex_array(ha):
   s = str()
   end_pos = 2

         while end_pos <= len(ha):
                  value = ha[end_pos - 2 : end_pos]
                  # convert a value (2 chars in hex) to int, to a ascii-char
                  s += chr(int(value, 16))
                  end_pos += 2
         return s

print 'fuse.mnt: %s' % decode_hex_array(fuse_mnt)
print 'fuse.subfs: %s' % decode_hex_array(fuse_subfs)
print 'FSDEF_netfs: %s' % decode_hex_array(FSDEF_netfs)
```

The above program should return the volume id, the subfs (glusterfs), and the netfs name, ie.. fuse.

#python /tmp/decode-keys.py

fuse.mnt: e545e1ea-e932-45ad-a003-2e91bacea403
fuse.subfs: glusterfs
FSDEF_netfs: fuse

5.8. Check if the data is cached on the local storage

Check if the cache back-end has the structure created for the new files we read now.
A structure similar to the one shown in the figure below should have been created.

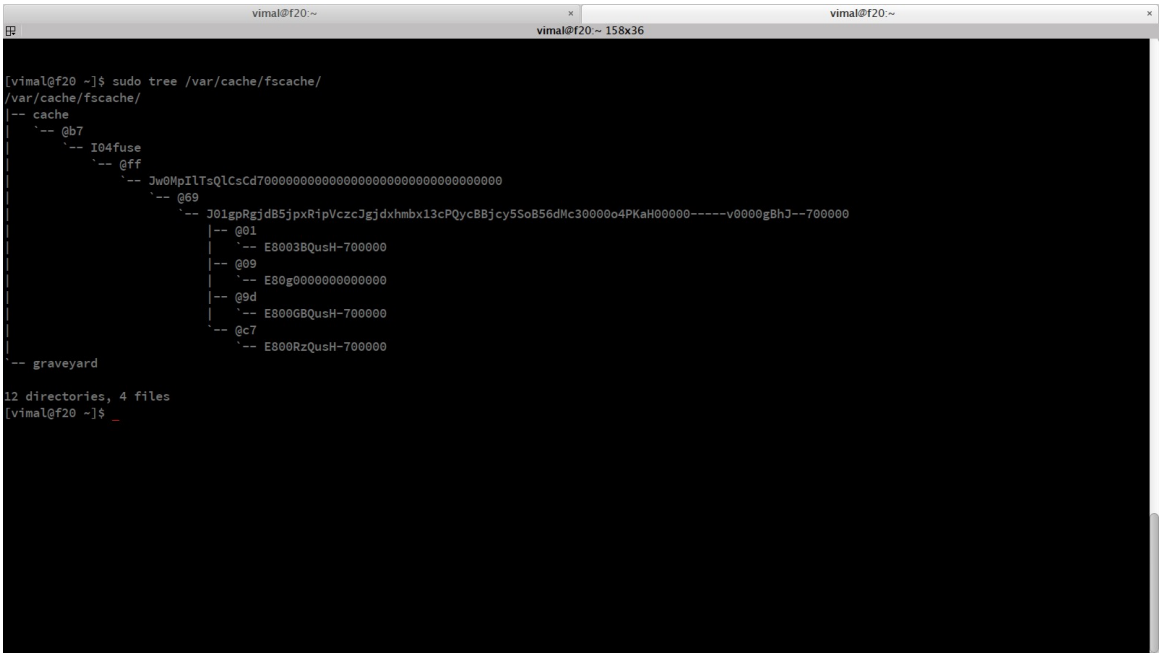

Fig 5.5 – Listing the cache contents at /var/cache/fscache/

The listing above shows the main index of the netfs (fuse), I04fuse. This is the name (fuse) which we defined in the 'fuse_cache_netfs' struct in section 4.3.

The entries in the cache structure starting with 'J' are for 'index objects' and stands for the mount point folder. The entries starting with 'E' stand for the data objects and contains the data which we read from the files.

Chapter 6 – Limitations, and future improvements

The current implementation of FSCache support in FUSE is in its very early stages. There are quite a few improvements needed to get this feature to be fully fledged. Some of them are:

a) Caching based on the GlusterFS volume-id, also called UUID. For now, we only have support for a single volume. This can be seen as a limitation. Mounting a second volume may either over-write the cache or corrupt it.

This is because we have support for only a single cookie in the 'fuse_mnt_cache_index_def' index.

This feature would need code additions in both glusterfs and fuse. FUSE would need to request a unique id for each volume from glusterfs, and glusterfs should return the requested id properly. This should be done by the glusterfs and fuse community.

b) Use of the glusterfs FSID rather than the fuse inode. Even though the fuse inode (from struct fuse_inode) works fine, a much more cleaner way would be to use the fsid.

c) Enabling auxiliary data caches which caches attributes of the data being cached.

d) Extend fuse to support a wide variety of FUSE-based file systems to use FSCache.

e) Enabling support for multiple glusterfs volumes simultaneously. As of now, only one glusterfs volume can be cached.

These are some of the features that needs to be implemented in the future. Further actions should be decided upon the feedback from the upstream FSCache community, after this is submitted to the developers.

Chapter 7 – Conclusion

Caching mechanisms have a lot to contribute to faster data access times. Common examples of caching are L1/L2 processor caches, and kernel page caches. But these are limited to a small size in capacity. FSCache is a service that can act as a dedicated caching utility, and can cache much larger data sizes.

Unfortunately, using FSCache is not straight forward. The file system that has to be cached has to have its driver tweaked to work with FSCache.

This project intended to create a working prototype for supporting FSCache in FUSE module, especially for GlusterFS file systems. FSCache has an API called 'netfs API' which helps to link it with the file system driver. A fully efficient prototype would take more time to get implemented, and cannot be completed in a period of three to four months.

My mentor, Niels, had the following to say about this project:

"While working on implementing support for FS-Cache in FUSE, it became evident that the FS-Cache and 'cachefile' components are not very developer friendly. The documentation is relatively good, but implementation details and how to apply the API is not always straight forward. Initial and simple tests that exercises the FS-Cache API will need to be restricted to initializing different indices and other routines for preparation.

A minimal but stable working prototype requires almost all the FS-Cache requirements to be implemented, instead of a planned specific subset. The proposed project would implement only minimal functionalities, but it seems that this is not an option for FS-Cache. The internals of FS-Cache expect a feature complete implementation, missing routines will cause stability issues, like extreme high loads, hangs and kernel panics and similar crashes.

Therefore the project became much bigger than initially scoped. A full implementation requires reviewing, understanding and modifying many of the internal workings of the FUSE Linux kernel module.

The additional knowledge needed includes, but is not restricted to how FUSE uses synchronous and asynchronous callbacks to facilitate communication between the Linux kernel and userspace, the Linux Memory Management subsystem that is used to fill and retrieve memory pages, and several of the internals of the Linux Virtual File System.

Most of these components have little relation with FS-Cache itself, but making the modifications in the FUSE kernel module requires a very good understanding of all surrounding code and how it is intended to work."

This feature would need more sub-features to be implemented to get it working as a full stable unit. As of now, even though the caching is working, there are chances that stability problems such as hangs, kernel panics etc.. can occur.

As such, this project is a stepping stone to the full fledged FSCache support in FUSE. Further development would need inputs from the upstream developers, and end-users, and for that, the current implementation would soon be submitted to the upstream community for review.

References

[1] The GNU General Public Licence:
https://gnu.org/licenses/gpl.html

[2] The Linux kernel and operating system
https://en.wikipedia.org/wiki/Linux

[3] What is Open Source, a description.
https://en.wikipedia.org/wiki/Open_source

[4] POSIX defined .
https://en.wikipedia.org/wiki/POSIX

[5] Top500 super computers
https://en.wikipedia.org/wiki/TOP500

[6] The RPM package manager.
http://www.rpm.org/

[7] The Linux kernel source rpm for version 3.16.6-200
http://kojipkgs.fedoraproject.org/packages/kernel/3.16.6/200.fc20/src/kernel-3.16.6-200.fc20.src.rpm

[8] The Fedora project.
https://fedoraproject.org/

[9] Qemu KVM.
http://wiki.qemu.org/KVM

[10] The virt-manager interface for KVM
http://virt-manager.org/

[11] The git source code manager.
https://en.wikipedia.org/wiki/Git_%28software%29

[12] Yellow dog updater modified (yum)
https://en.wikipedia.org/wiki/Yellowdog_Updater,_Modified

[13] Cscope – The tool for browing source code
http://cscope.sourceforge.net

[14] Fedora 20 installation ISO
http://download.fedoraproject.org/pub/fedora/linux/releases/20/Live/x86_64/Fedora-

Live-Desktop-x86_64-20-1.iso

[15] The fedora 20 installation documentation.
https://docs.fedoraproject.org/en-US/Fedora/20/html/Installation_Guide/index.html

[16] Git and source-code branching
https://www.atlassian.com/git/tutorials/using-branches/

[17] The Virtual File System (VFS)
https://en.wikipedia.org/wiki/Virtual_file_system

[18] A list of file systems using the FUSE kernel module
http://sourceforge.net/p/fuse/wiki/FileSystems/

[19] GlusterFS software storage solutions
http://gluster.org/

[20] David Howells, "FS-Cache: A Network Filesystem Caching Facility".
http://people.redhat.com/~dhowells/fscache/FS-Cache.pdf

[21] FSCache netfs-api documentation.
https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/filesystems/caching/netfs-api.txt