

Filesystem Interface for the Git Version Control System

Final Report

Reilly Grant <rm@seas.upenn.edu>

Adviser: Jonathan M. Smith

April 24, 2009

Abstract

Figfs (the Filesystem Interface to Git FileSystem) provides a new way for developers to interact with their version control repository. The repository is presented as a filesystem which allows multiple versions and branches of the project code to be viewed simultaneously and without the need to reconfigure the user's workspace. It is also a more natural interface as existing tools can be used without being made aware of the versioning system in place. Presenting branches and history in this fashion will increase developer productivity and encourage the use of branches for managing feature development.

This project targets Git¹, a distributed version control system, and expands on the work of gitfs[11] (the Git FileSystem), which only provides read-only access to the project repository. Figfs allows the instant creation of new writable workspaces without the need to create any new files on the local filesystem. The goal of this project is to make figfs the preferred environment for all development tasks.

Introduction

Version control systems (VCS) are a critical part of the software development process. They provide a way to manage what code is part of a project and track changes to it. While traditionally used to manage source code, they have also been adapted to store system configurations, websites, documents and other data files. Without a version control system software development would be chaos.

When searching for the cause of a problem it is often useful to look through the previous versions of a project in order to find the change that started it, and who made it. Traditionally, there are two ways to view a previous revision of a file from a VCS. One is to revert the entire copy on the local filesystem back to the previous revision. This, however, disrupts any

¹The meaning of the name "Git" is not standardized but may refer to the ego of Linus Torvalds, it's creator, or stand for "Global Information Tracker".

files which are currently being edited. It is also a potentially enormous I/O operation since it must rewrite every file that changed between revisions to bring the files to a consistent state. The other method is to request the text of a single file using the VCS utility program. Often this is what happens when the user clicks on a revision in a VCS GUI application. The file is saved to a temporary directory and an application is invoked to view it. This is usually the more convenient option, since the user often only wants to examine a few files. It is far from ideal, however, because the files are not being viewed in the context of the version they belong to, preventing more advanced editors, such as IDEs, from presenting the file as belonging to a larger project.

This project constructs a new filesystem, called *figfs*, in which previous revisions of the entire repository can be viewed as if they existed on the local filesystem. In addition to a read-only historical view the system allows a filesystem-backed view to be used for normal development and compilation. This will allow for better parallel development of features and fixes, by allowing the user to quickly start work on whatever branch or revision he wishes. It will also be more efficient because the filesystem mediates, and is able to track, file modifications. The system can then avoid scanning the entire directory structure for files that have been changed and should be committed as part of the next revision.

Related Work

A plethora of version control systems exist, and they fall mainly into two categories: centralized and distributed. In a centralized VCS all revisions and meta-data are stored on a central repository server. Users may *check out* one particular version and *commit*, sometimes called *submit* or *check in*, new revisions if they have the proper access. In a distributed VCS there may still be a central repository which contains the official project history, but every developer has his own copy, or *clone*. A developer can commit to his own copy whenever he likes, and at some point may choose to *push* these commits to the central repository, or to other developers. He can also *pull* new revisions from other repositories into his own. Distributed systems have the advantage of not creating a bottleneck for users requesting information from the repository and allow users to interact with the repository when they are disconnected from the network. With either type of system on the developer's machine the VCS creates a *working copy*, a directory of files which reflects the state of the repository at one particular revision. The developer can edit this copy and create a new revision based on those changes, or can *update* the working copy to reflect the state of the repository at any other revision, forward or backwards in time. Meta-data, such as revision logs, can be viewed using the VCS's included utility program.

Beyond the basic version management functions a VCS creates the ability to *branch* and *merge* lines of development. Branching creates two copies of the code-base, allowing multiple features to be developed in parallel and for versions in different parts of the release cycle to progress independently. Merging then allows those pieces come back together and share new features and bug fixes. Systems such as Subversion[8] and Perforce[14] do not track the concept of a branch per se, but recommend a directory structure where branches exist in

parallel to one another and changes can be merged between files. The history of merges from one file to another is maintained separately from the history of that particular file. Others such as Git[2] and Mercurial[13] explicitly store a branch as a separate chain of commits over the same collection of files. The branches are tagged with separate names instead of being in separate directories. A single working copy is sufficient for some development, but, especially when tracking bugs or managing releases, it becomes necessary to browse more than one at a time. In the former type of system the user can check out a single directory tree which logically contains more than one branch. In the later type, however, two branches cannot be checked out side by side without making a separate clone of the repository. In either case, accessing another branch or historical revision requires writing the contents of the entire revision to the local filesystem. In contrast to these systems, figfs gives instant access to any version. The storage requirements for keeping this data available is negligible because of the tightly packed on-disk format of a Git repository.

The ability to access past revisions in a repository via the filesystem has been implemented before. Gitfs and svnfs[12] (which is the same as gitfs except that it uses Subversion) implement a read-only view of repository history. The advantage of gitfs over svnfs is that Git is a distributed system and thus maintains a copy of the entire repository on the local machine, eliminating network lag when fetching revisions. A commercial system, Rational ClearCase[9], offers a writable filesystem view of the repository, MVFS (MultiVersion File System), as an alternative to checking out files to the local filesystem. As with svnfs the performance of this system suffers from the need to query over the network for uncached file data. Figfs eliminates this problem because a Git repository is stored entirely locally.

Technical Approach

In order to provide a filesystem service figfs uses the Filesystem in Userspace (FUSE)[1] library to interface a user-space filesystem daemon with the kernel filesystem code. This library has been used by a number of projects, such as gitfs and svnfs, to present information through the filesystem in novel ways. This also allows figfs to be written in a higher level language (OCaml) instead of C. The diagram in Figure 1 shows how FUSE manages communication between the filesystem implementation, the kernel and another user-space application.

FUSE consists of two parts, a kernel-space driver and a user-space library. The kernel driver fills in the required Linux filesystem interfaces and provides a `/dev/fuse` device for communication with the user-space component. The library abstracts the communication protocol over this interface into a C API that resembles the standard filesystem interface (`open`, `read`, `write`, `readdir`, etc.). To write a filesystem in OCaml a fork of the OCamlFuse[7] library is used. OCamlFuse provides a light wrapper around the C API allowing the C callback functions expected by the FUSE library to be implemented by OCaml functions. While a multi-threaded mode is supported, it is not as efficient as single-threaded operation due to thread creation overhead. In the future a more purely OCaml version of this library may be written to reduce the dependency on the interface generators used by the original

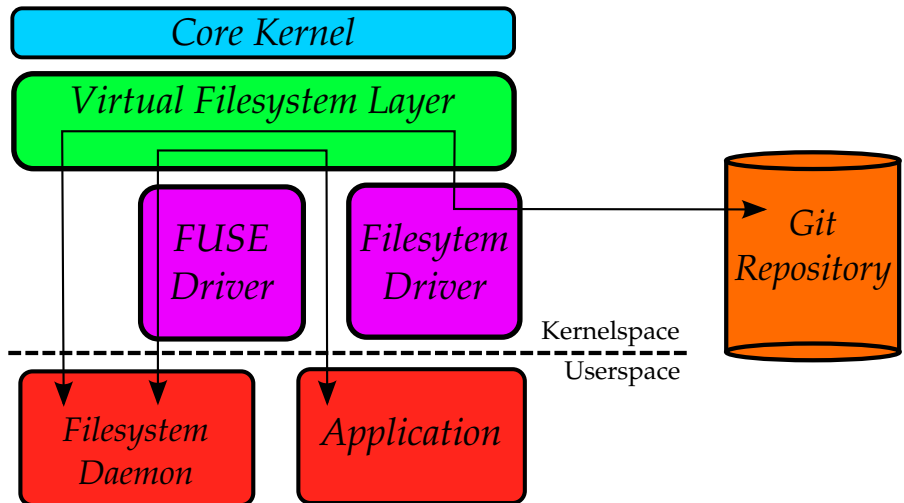


Figure 1: A FUSE application allows a filesystem to be implemented as a user-space process. An application’s request is passed through the VFS layer and into the FUSE driver which sends the request to the userspace filesystem daemon. Figs then accesses the Git repository through the normal filesystem and returns the resulting data to the application.

OCamlFuse author.

A Git repository consists of an object database and some pieces of meta-data, including tags and branches. Objects in the object database are of one of three types, commit nodes, tag nodes, tree nodes, and blobs. A commit node represents a committed revision and contains pointers back to its parent commits. It may have more than one parent if it represents a merge between two or more branches. A tag node points to a single commit and serves to mark the commit as a milestone or released version. A tree node represents a directory in the filesystem and contains a list of files, their permissions, and pointers to blobs. Blobs contain file data. Each node is identified by the SHA-1[6] hash of its contents and is either stored as a file named as such, or in a packed archive with other objects, indexed by its SHA-1 hash. Files and directories which are not changed are not stored twice because the object database is content addressable. The cryptographic security of SHA-1 ensures that the identifier of an object can be used to verify its integrity along with the integrity of all the objects it references. The contents of a Git repository can be pictured as a directed acyclic graph. An example is given in Figure 2. Figs processes this structure to access the repository.

On disk objects are stored in one of two formats. When first created an object is stored in its own file named by the SHA-1 hash of its contents and compressed with the DEFLATE[10] compression algorithm. This is called the “loose file” format. These files are easy to create, but over time they would take up too much space on the filesystem as each file each requires its own i-node, is often smaller than a block and stores a complete copy of the object. This

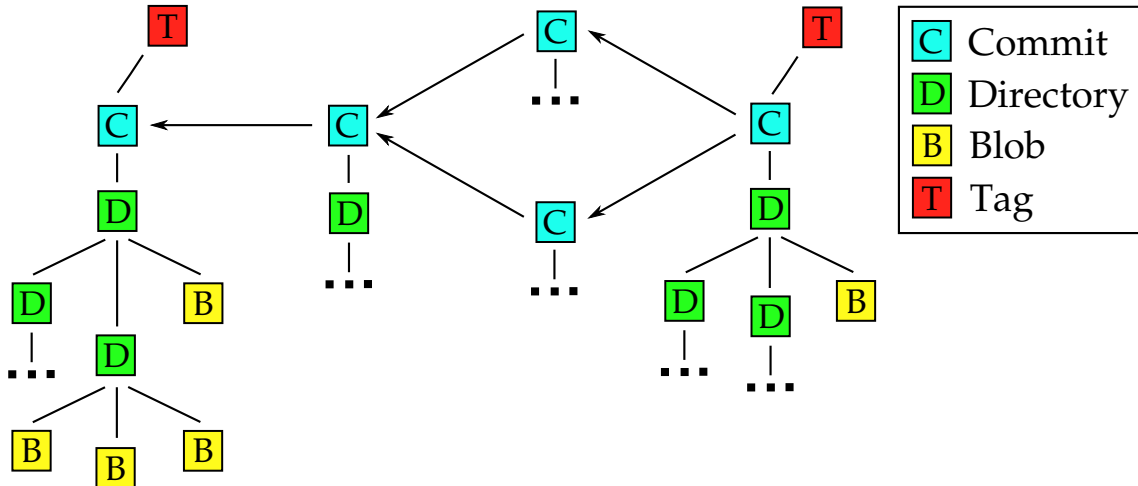


Figure 2: An example Git repository depicted as a directed acyclic graph. If two files or directories contain the same data then the data is not duplicated. Instead they are both represented by a single node.

many files would push the scalability of the native filesystem implementation.²

When transmitting objects over the network and storing older objects the “packed file” format is used instead. First, multiple objects are stored in a single file, relieving stress on the native filesystem. An index file is used to map object identifiers (still SHA-1 hashes of the object data) to locations in the pack file. Objects are sorted in the pack file by a magic heuristic³ which optimizes for storage locality and compression efficiency. Objects are then delta-compressed such that generally only the difference between each subsequent revision is stored. The sorting heuristic guarantees that the newest revision of a file is stored completely while previous revisions are stored as reverse patches against it to optimize for the common case of accessing the most recent revision. This data is then also compressed with the DEFLATE algorithm.

Most operations performed by the git utility involve processing each object only a few times. In contrast, a filesystem is frequently asked about the same path multiple times. For example, accessing a file in a sub-directory requires querying all intermediate directories to check for permissions to traverse them. Therefore, to increase performance a caching layer has been introduced between the object lookup functions and the repository access functions. Since objects are identified by their SHA-1 hash a simple string lookup hash table is suggested. Also, since Git assumes that there will be no collisions between object identifiers, once an object is cached, it never needs to be checked for freshness.

²For example, the Linux kernel repository contains 978,592 objects as of December 1st, 2008. Unpacking these objects into loose files took over 1 hour and consumed 6.4 GB of disk space, compared to the 252 MB used to store the single packed file.

³See `Documentation/technical/pack-heuristics.txt` in the Git source distribution for a discussion by Linus Torvalds of the reasoning behind the heuristic.

Figs presents a number of views over the repository history. Each view attaches a particular commit to a directory under the mount point. The `/commit` directory allows direct access to a particular committed revision. The `/branch` directory contains a directory for each branch. The `/tag` directory contains a directory for each tagged version. Under each of these directories is a complete view of the state of the repository at that revision. These copies are read-only because they refer to a historical commit. The `/workspace` directory, however, is different. The user can define a workspace from a given commit and writes to this workspace will be allowed.

In order to implement a writable workspace the system maintains a write cache directory for each workspace and a database of files modified for each workspace. The database is stored using SQLite[5] and stores only file meta-data. The SQLite engine is useful for projects such as this because it combines a pre-made embeddable storage engine with a standard interface for queries. Files in the write cache are stored on the native filesystem. When a file is written it is dumped into the write cache and further reads and write go directly to this file. When a file is deleted the file is marked in the database as deleted and is thus omitted from directory listings even if it exists in the underlying revision.

Operations such as creating a workspace require communications between the user and the filesystem daemon. In gitfs this is accomplished by performing IPC calls over a Unix socket. In figs the need for a socket is removed by creating a special file available in all directories within the mounted filesystem. When a command is written to the file the associated action is performed by the filesystem daemon. This method also allows the daemon to detect which directory the command was executed in by the path used to access the control file.

The write cache also can allow the system to detect modified files without scanning the entire workspace. In the future this ability will be used to optimize the way that changes are committed back into the repository. After committing a new revision the workspace will refer to this new commit.

Technical Challenges

The first great technical challenge so far has been implementing support for reading objects out of the Git repository. Because of the size and complexity of the Git code base, the author chose not to use the OCaml foreign function interface to leverage existing code from the Git project. By implementing this support in OCaml the system is in fact less complex. Because Git is still young and its core was developed by a single developer, however, not all of the on-disk format is well documented and somewhat obscure. For example, there are at least two custom binary integer representations used in pack files. OCaml also does not have a built-in implementation of, or bindings to, the zlib[3] library for DEFLATE compression, so a bindings had to be created for this project. After some later difficulties with the custom zlib bindings the project was switched to using camlzip[4], a more stable and complete implementation.

With this foundation in place and well tested, however, the initial implementation of a

read-only filesystem on top of it proved to be quite simple. The author attributes this to the conciseness of expression and ease of prototyping in the OCaml language.

Implementing the writable workspaces presented the technical challenge for the second phase of the project. Implementing a read-only filesystem requires writing only a few system calls. To make that system writable, however, requires more than implementing `write()`. Applications expect to be able to modify permissions, create directories, symlinks, update modification times, etc. Each of these required a careful reading of the API specification and translation into operations on the repository and write cache. The second phase is now complete, users can create, edit, and delete files within a workspace. The ultimate test was to be able to build figfs from its own Git repository in a workspace mounted through figfs itself.

Future Work

In future versions figfs will support more of the functionality of a traditional traditional working copy. To do this some features of the Git utility will need to be reimplemented in the filesystem daemon. To avoid reimplementing large pieces of functionality the system will attempt to have each workspace appear to the Git utility as if it were a normal checked out copy created by the utility. This will require researching the behavior of the utility and its expectations of the environment.

Once the solid foundation of these features is completed there are a number of additional projects that can be implemented on top of the basic system. For example, more functions of the VCS utility such as displaying repository metadata and revision logs can be exposed directly through the filesystem. Besides being used for source control this filesystem can also be used to implement a feature similar to Apple's Time Machine. If one uses this filesystem to store their home directory a periodic script can create new commits in the background and old versions can be accessed through an interface similar to NetApp's `.snapshots` folder. For different uses it may be more intuitive to have the basic filesystem layout to look different. Thus, future versions will allow the layout to be reconfigured to suit the needs of the application.

Conclusion

By introducing version control at the filesystem level it becomes accessible to every program on the system. The goal of this project was to extend that beyond the read-only views available in current version control based filesystems. The workspace functionality gives the developer the flexibility to switch rapidly between different source branches or quickly create a workspace to test a feature and then get rid of it when no longer needed.

To some people the most shocking design decision in this project was the choice of language, OCaml. There have been advantages and disadvantages of this choice. Since this is a systems level project it must interface with low level system libraries. These are written

in C and finding good bindings to them has been a challenge. If not in OCaml then this project would have been completed in C. OCaml, however, provides all the flexibility and control of C without the need to do memory management and deal with the programming errors that that implies. In addition, native support for pattern matching and data structures such as lists make OCaml more suited for rapid development.

It is unfortunate that the Git project does not offer an interface for 3rd party programs to use their code directly. It seems that those systems which do interface with Git do so using the command line utilities. For this project it was felt that this interface would not be fast enough for use in a filesystem. From documentation and comments in the Git project it seems like some effort is underway to make a Git library available at some point in the future. For this project, however, reimplementing the functionality required to read the repository offered the opportunity to more deeply understand the way that Git works.

At this point, figfs is in an early beta state. The core feature set is in place to read and write data on the filesystem. In its current state figfs can be useful to developers. The project has been released under the open source GPLv2 license and is available for download by the public. Once a number of known bugs are fixed there will be a general announcement of its availability to the Git community at large. It is the author's hope that the project will be accepted by the community and attract users and other contributors to the project. Development will continue after the completion of Senior Design.

References

- [1] FUSE: Filesystem in Userspace, September 2008. <http://fuse.sourceforge.net>

FUSE provides a kernel driver and library for implementing filesystems as userspace processes under Linux and other Unix systems. It has been used by other projects, such as gitfs and svnfs, to provide novel filesystem interfaces to user applications.

- [2] Git User's Manual, September 2008. <http://www.kernel.org/pub/software/scm/git/docs/user-manual.html>

Git implements a high performance distributed version control system. It was designed single-handedly by Linus Torvalds after BitMover Inc. revoked the license it had previously offered to Open Source projects to use its BitKeeper VCS. Since then it has become the primary tool for Linux kernel development and has also been adopted by other Open Source projects. Git repositories are stored locally and highly compressed which makes it an ideal backend for this project.

- [3] zlib, September 2008. <http://www.zlib.net>

Zlib is the standard implementation of the DEFLATE algorithm present on most Linux systems as well as other Unix variants.

- [4] camlzip, April 2009. <http://cristal.inria.fr/~xleroy/software.html#camlzip>

CamIzip provides a simple OCaml binding for the zlib library for DEFLATE compression.

- [5] Sqlite, April 2009. <http://sqlite.org>

SQLite provides an embeddable and lightweight storage engine perfect for storing information in projects such as this.

- [6] D. Eastlake 3rd and P. Jones. *US Secure Hash Algorithm 1 (SHA1)*. Network Working Group, September 2001. *Standardized as RFC3174*

The SHA-1 hash is a cryptographic hash selected by the NIST. While its security has been recently questioned it remains mostly unbroken and is used by Git to name content addressable objects.

- [7] Vincenzo Ciancia. OCamlFuse, September 2008. <http://ocamlfuse.sourceforge.net>

OCamlFuse provides OCaml bindings to the FUSE library. The project is mostly unmaintained, the code used by figfs is a minor fork.

- [8] B. Collins-Sussman, B. W. Fitzpatrick, and M. C. Pilato. *Version Control with Subversion*. O'Reilly & Associates, Inc., Sebastopol, CA, 2008.

Subversion is one of the most popular Open Source version control systems. It was created to be the successor to CVS and is similarly centralized. Subversion was designed to keep the CVS interface while fixing some of its issues with storage and scalability.

- [9] Rational Software Corporation. *Rational ClearCase: Administrator's Guide*. Lexington, MA, 2001.

ClearCase is a commercial centralized version control system. It implements a filesystem similar to this project, but is unable to achieve high performance because of its centralized repository.

- [10] P. Deutsch. *DEFLATE Compressed Data Format Specification version 1.3*. Network Working Group, May 1996. *Standardized as RFC1951*

DEFLATE is a standard stream compression algorithm. It is used by Git to compress objects on disk.

- [11] Mitchell Blank Jr. `gitfs`, September 2008. <http://www.sfgoth.com/~mitch/linux/gitfs>

This project was the inspiration for this work. While first released in June 2005, it seems to have been abandoned since, with the last release in December 2006. Its webpage indicates that there are many features that the author has yet to implement. Figfs will implement all the features of this read-only filesystem, as well as the new workspace features introduced in this paper.

- [12] John Madden. SvnFs, September 2008. <http://www.jmadden.eu/index.php/svnfs>
Another project similar to gitfs, svnfs (also currently unmaintained) is based on the Subversion VCS. Because of Subversion's centralized nature it must reach out over the network for all repository information.
- [13] Bryan O'Sullivan. *Distributed revision control with Mercurial*. 2007. <http://hgbook.red-bean.com/hgbook.pdf>
Mercurial is a distributed version control system similar to Git. It was not chosen for the Linux kernel because Linus Torvalds claimed it was not high performance enough for his needs. This is probably because it is implemented in Python instead of C.
- [14] Perforce Software. *P4 User's Guide*. Alameda, CA, 2008.
Perforce is a commercial grade version control system used at many companies including Google and Microsoft. It has a centralized architecture and management features that make it especially useful in a corporate environment where tighter control over user access is required.