Fossilizing NetBSD: The road to modern version control

Jörg Sonnenberger <joerg@NetBSD.org>

October 3, 2011

Abstract

This paper discusses cvs2fossil, a new repository conversion tool for CVS. It gives an overview of how different CVS features are handled and discusses some issues found in real world repositories. The second part analyses the performance of Fossil and compares it to other VCSs.

1 Introduction

The NetBSD project has successfully deployed CVS for over one and a half decades. The main modules, pkgsrc and src, provide a huge challenge for any replacement. The pkgsrc module challenges scalability by having over 60,000 files per working copy and a total of over 100,000 files in the repository. The src module challenges scalability both in terms of raw size (4.3GB of RCS files) and the large history of 240,000 change sets.

Over the last three years a number of attempts to provide conversions to modern version control systems (VCS) have been made. The different VCS and the associated conversion tools all have different shortcomings and no clean consensus could be reached to move into one direction or another.

One important tool is Simon Schubert's fromcvs. [1] It was the only option for continously replicating the CVS changes into Git without breaking the normal updating process of the target VCS. This raised the question of how much work a RCS/CVS conversion tool would be that fits the requirements of NetBSD:

- Be faithful: honour RCS keywords.
- Be smart: properly deal with vendor branches and magic CVS revisions.
- Be fast: finish in much less than a day on reasonable modern hardware.
- Be helpful: provide support for cleaning up the mess that a large scale repository ends up being.

At the time concrete plans started to form, Richard Hipp had started making Fossil VCS project [2] more visible and managed to cut the legalese associated with the source code by moving to a BSD-like license. The result is attractive–a compact binary under a liberal license with few external dependencies and a

[1] `http://ww2.fs.ei.tum.de/~corecode/hg/fromcvs`

[2] `http://www.fossil-scm.org`

fitting name. A project was born: converting the NetBSD repository to Fossil and evaluating the scalability.

The first part of this paper discusses the resulting conversion tool cvs2fossil. This includes an overview of how different CVS features work and issues that have been found in the NetBSD repository.

The second part of the paper analyses the current performance of Fossil for various important operations and changes made in Fossil to deal with scalability limits. This includes a limited comparison to other VCSs.

2 Anatomy of a CVS repository

2.1 Overview

A CVS repository is a loose collection of RCS files. All files are independent of each other. Inside a RCS file, the commits form a directed acyclic graph. Each commit has a revision number (unique per file), a time stamp, the author, state and commit message associated. The state is either "Exp" or "dead".

Revision numbers normally increase by one between revisions on the same branch. Exceptions are the result of manual interference, e.g. using "cvs admin -o". Revision numbers consist of an odd number of small integers and each branch adds another two components. The first revision of a file is 1.1.

The content of the head revision is stored as full text. Revisions on the head branch are stored as backwards delta from the newer revision to the older revision. Branch revisions on the other hand are stored as forward delta from the older revision to the newer revision. This makes checking out the head very fast but obtaining revisions of an old branch relatively slow.

2.2 Tags and branches

CVS allows to associate symbol versions to individual revisions and whole branches. All symbols have an odd number of components with one exception. Vendor branch tags are special and will be discussed in detail in the next section.

Whether a symbol specifies a tag or a branch can be verified by looking at the second right-most component of the revision number. For branches this component is 0. The prefix of branch revisions is obtained by removing this component, e.g. the symbolic version "1.2.0.2" specifies the branch with the prefix "1.2.2". A tag on that branch would be "1.2.2.1", the first revision of it.

2.3 Vendor branches

Vendor branches are a special feature of CVS and are created as result of the "cvs import" command. They use the special "1.1.1" branch and a so-called default branches. A default branch instructs cvs to use the revisions on the specified branch instead of checking out the revision named as head. The first regular commit removes the default branch marker and uses the "1.2" revision number as head.

Vendor branches create two interesting problems. First, they are the only important case where multiple branch tags can share the same revision and there is no way to distinguish between them. This means that one revision on the “1.1.1” branch can be part of the multiple logical branches. Second, they make processing the head branch more complicated as the temporal order goes from “1.1” to “1.1.1.1”, follows the vendor branch until the last revision that is older than “1.2” and switches to “1.2” to follow the default branch.

“1.1” and “1.1.1.1” share the same time for vendor branches. This is not true if a file has been added via “cvs add” first and a “cvs import” has been done later.

The use of “cvs import -n” is strongly discouraged and can result in time inversions in the repository even without manual patching. That means that a revision has a date newer than a descendant revision.

2.4 Keywords

CVS supports keyword expansion for files stored in the repository. This is enabled by default, but can be suppressed on a per file basis, e.g. to allow storing binary files, or for specific operations. The list of keywords supported by CVS includes `$Id$`, `$Date$`, `$Log$`. Most of the BSDs use a local modification to get a custom keyword. NetBSD is using `$NetBSD$`, which expands to the same text as `$Id$`. FreeBSD historically used a custom tag that included the full module path. OpenBSD has a custom tag for manual pages.

Some tools dealing with CVS repositories don’t handle keyword expansion and use the literal value stored in the repository. This is normally seen as off-by-one error, e.g. the keyword contains the data from the last revision. The content of the unexpanded version can be arbitrary; CVS checks the literal content of the file in.

3 Anatomy of a Fossil repository

A Fossil repository is a SQLite database having both permanent and ephemeral tables. The ephemeral tables are derived meta data used to efficiently find and access the permanent data. From the permanent tables, only the “blob” and “delta” tables are relevant for the conversion process. The “blob” table contains all artifacts. The artifact is the fundamental storage unit in Fossil. It can either represent a specific version of a file, a commit or other objects for the wiki or ticket system. Artifacts are stored either as zlib compressed objects or as delta relative to another artifact. The “delta” table connects artifacts stored as delta with the corresponding reference. Artifacts are identified by their SHA1 hash.

Artifacts representing commits are called manifests. Manifests are simple text files and include all data to describe a commit like author, commit date, a list of files and SHA1 hash of the content of each file. Originally manifests always included a full list of all files. This works well for small to medium sized trees. It does result in very large manifests for trees the size of NetBSD src though. During the “rebuild” phase where the ephemeral tables are populated, all manifests have to be parsed. This has been addressed by introducing baseline manifests. If a manifest references a baseline manifest, it inherits the full file list of that baseline. The baseline manifest itself must not use this feature itself.

4 Conversion from CVS to Fossil

4.1 Overview

The conversion process has to manage a lot of data. For the NetBSD src tree, 190,000 files with over 1 million revisions have to be aggregated to 240,000 change sets. The total size of all uncompressed revisions is 24GB. Using a SQL engine to access this simplifies the code by doing the heavy lifting and indexing in a higher level language.

The conversion is split into consecutive phases:

1. Import from CVS into a SQLite database; compute branch points and the mapping between revisions and branches.

2. Handle vendor branches and check consistency of branch points.

3. Computation of branch creation times.

4. Compute manifests and copy referenced blobs into target repository.

The phases are decoupled to allow manual changes for sanitation. This can be used to adjust branch points or remove undesirable branches completely.

4.2 Import phase

This phase deals with parsing the RCS files and expanding all revisions. Keywords are handled as if “cvs update” was used to fetch each revision explicitly. The content of all revisions is compressed to save space and deltas for adjacent revisions are computed in the background.

The content and the meta data of all revisions is not the only information computed and stored in this phase. The relationship between revisions (ancestor and descendent) is saved as well as all tags. The first pass for assigning revisions to branches is done. Branch points are derived and revisions to skip memorised.

A branch point is the revision a branch is attached to. It can be obtained by stripping the last component of the branch revision prefix. If the first revision of a branch is in state “dead” and has the same commit time as the branch point, it means that the file was not initially part of the branch, but added later via “cvs add”. To handle this, the branch point is explicitly removed in this case and the “dead” revision added to the list of skipped revisions.

Old repositories tend to accumulate cases where this rule has been violated. Since branches and date based checkouts are already problematic in CVS, this is often not visible. It does create problems for the third phase, the branch creation time though.

One issue that is relevant for this phase is that “cvs commit” sometimes duplicates the revision, e.g. two consecutive revisions have exactly the same meta data and the content only differs in the keyword expansion. This can be detected by checking the strict monotony of commits.

4.3 Vendor branches

The vendor branch processing deals with a number of separate but related issues:

1. If there is more than one vendor branch on a file, one of them has a branch point. Duplicate this branch point for the other vendor branches on the file. This deals with two "cvs import" calls for different vendor branch names.

2. Warn if one branch is spawned from two different parent branches.

3. Weave the vendor branch revisions into the chronological order of the default branch as needed.

4. Add "1.1" to the skipped revisions if it has been created by "cvs import".

5. Add "1.1" to the skipped revisions if it is "dead" (e.g. the file was initially added on a branch).

Issues for this phase range from missing default branches to inconsistent branch relationships. Missing default branches are a phenomenon observed in pkgsrc where files have been created by "cvs import" and have a head revision of "1.1", but no default branch. Side effect is that checkouts would extract this version and not "1.1.1.1", but this fact gets lost on the next commit. The only way to detect this problem afterwards is by looking for tags on "1.1". Since it is relatively rare and the conditions are not well understood, this isn't handled automatically.

Inconsistent branch relationships most often happen with partial branches (branches that cover only part of the tree). Sometimes additional files are branched at a later point, ignoring the correct relationship. This is not relevant for CVS. Since there is no clear way for automatically handling such conditions, a warning is issued.

4.4 Branch creation times

This phase tries to synthesize the time when the branches were created. It has three sets of dates to consider for this purpose:

1. The date of the branch point.

2. The date of the next revision on the parent branch.

3. The date of the first revision on the branch itself.

Consistent branching would ensure that no commit happens between the branch point of a file and the branch creation time. All branch points and the first revisions on the branch should not be later than the branch creation time.

If all branches are created from consistent tags or it has been ensured that branch creation isn't disturbed by commits, all three sets of dates are consistent. Selection one set matters, if branch creation is not atomic.

Two major approaches exist for dealing with that:

1. Create the branch as early as necessary and replay changes on the branch. This is the approach taken by cvs2svn.

2. Create the branch at the time of the last branch point and backout all accidental changes. This is the approach taken here.

The cvs2svn approach has the advantage of preserving small changes and allows more useful annotations. It is quite a bit more complex though. Avoiding this complexity can be justified by:

- A complete tagging history from CVSROOT/history can provide precise time stamps from an external source. This can easily be hooked up into the conversion process.
- Most branches with inconsistencies are the result of incorrect "cvs add" operation. This was the case for the analysed NetBSD repositories as well as the Tk repository.
- Proper handling of such "cvs add" operations require human intervention if no history is present. The number of affected case tends to be small and the gain is obvious. One time use can be as simple as executing a DELETE statement after the first import phase.
- A clean up process can be scripted to insert a "dead" revision using "rcs" and "ci". This approach has been used on the NetBSD repository.
- Post-processing after the import can identify the problematic change sets. One example is the "fossil test-timewarp-list" command.

The simple approach cuts down the number of artificial commits and therefore results in a cleaner repository.

4.5 Fossil import

The last phase combines the data of the previous phases to compute the actual manifests. This phase orders all versions of a given branch by the commit date and path. Change set aggregation is done based on the following rules:

- All commits to be aggregated have the same author and the same commit message.
- A file is changed at most once in a change set.
- The creation time of a child branch is not crossed.
- The difference between the oldest and newest commit is no larger than 10 minutes.

Order by path as secondary criterion makes the order stable. This allows repeating the same process with a newer repository and getting a compatible result. This is the poor man's version of incremental conversion.

One special case is a branch that doesn't contain any changes. This is the only case where cvs2fossil creates an artifical manifest. If there is a change, the branch tagging is packed into the first manifest of this branch.

VCS	Repository size	IO for clone
Fossil	2221MB	59KB out / 1384MB in
Git	717MB	2.1MB out / 651MB in
Mercurial	1848MB	893KB out / 3584MB in (compressed: 451KB in, 910MB out)
CVS	4286MB	n/a

Figure 1: Repository sizes

4.6 Performance

The run time for the conversion process on an Opteron 1389 (Quad-core, 3 GHz) with a RAID-1 of two 7200rpm SATA disks with 4GB memory is as following:

CVS import	34min
Vendor branches	4min
Branch creation time	24s
Fossil import	3h 56min

This does not include the time for “fossil rebuild” since the continous update process doesn’t actually run it. The resulting repository is pushed to a local master copy and synchronised to the public repository from that.

Reports indicate that cvs2svn needs at least twice the time for larger repositories.

5 Fossil performance

The most common operations for developers are creating a new working copy, looking at the changing in a working copy, committing and updating. Push and pull operations are not analysed in detail since they tend to be network bound. The case of network operation is also outside the scope.

All test cases involve the converted CVS repository and the default branch. For this purpose the Fossil repository has been converted to Git and the Git repository imported into Mercurial. The numbers are average of six runs with hot caches.

Creating a new working involves either “fossil open”, which creates a checkout associated with the local repository file, or cloning the repository from a local or remote destination. Figure 2 shows that Fossil has somewhere between Git and Mercurial for this. The majority of the work is writing all the files. Fossil currently spends quite a bit of time before actually writing all files, further work is needed to investigate exactly what is going on.

Figure 3 shows the necessary time for finding all changes in the working copy after applying a small patch that changes two files. This operations involves comparing the modification time of all files in the tree with the recorded version. After the “status” command, the “diff” command is used to create a unified diff.

Figure 4 shows the time for committing the changes and updating to the parent of the original head version. At least “fossil update” is doing too much work. It doesn’t fully use the meta data tables yet and has to fully parse the manifests of old and new version. There is room for improvements here. The long time for “fossil commit” also has to be investigated.

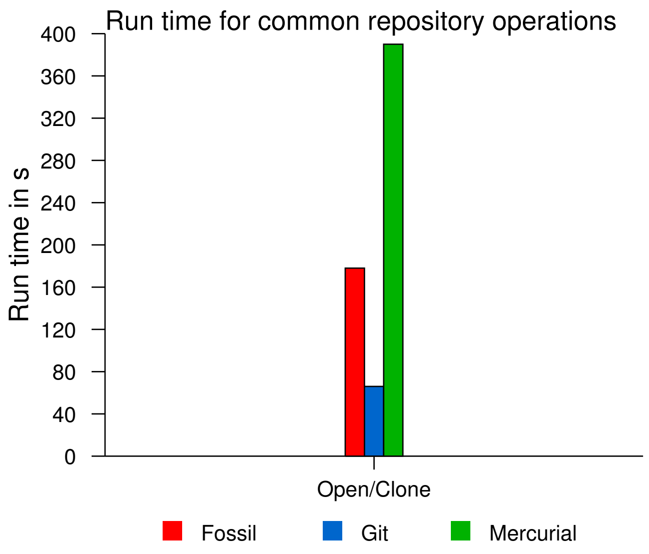

Figure 2: Time for creating a working copy

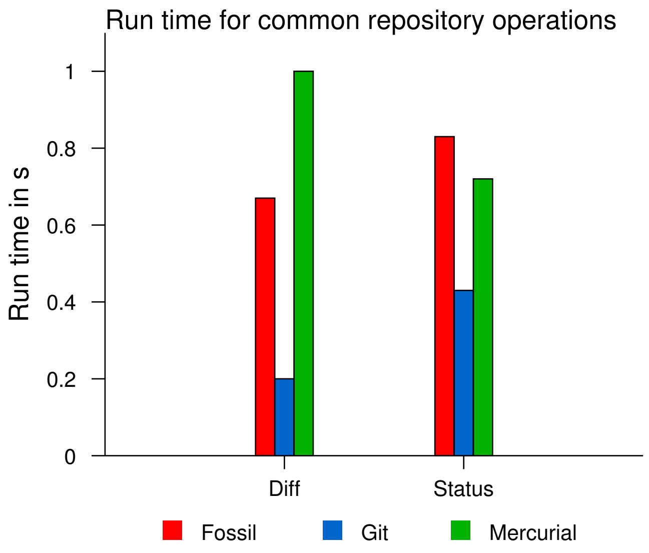

Figure 3: Time for finding changes in a working copy

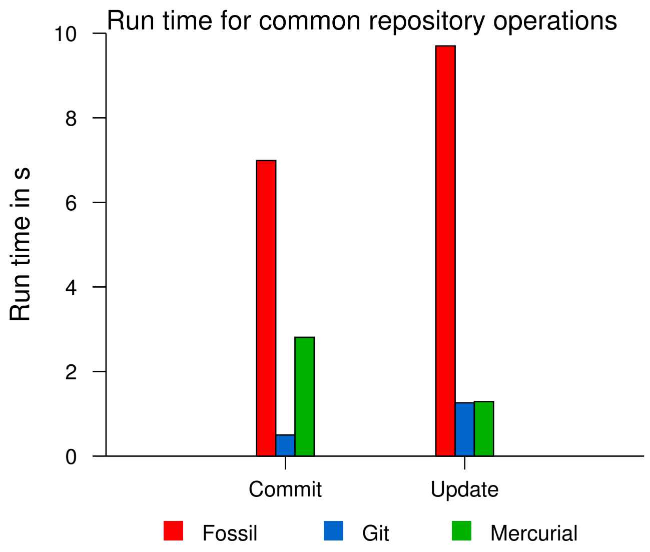

Figure 4: Time for committing changes and updating

6 Summary and future work

The new cvs2fossil tool provides a fast conversion tool that can handle large repositories. The number of artificial commits and conversion glitches could be minimized by cleaning up various issues. This was made easy by exploiting the database and writing small Python scripts calling “rcs” and related programs.

Various tests show that the Fossil repository and CVS give exactly the same output for individual revisions. Differences when comparing working copies of specific branches are accounted for.

The performance of Fossil is competitive. Some areas like “fossil pull” need further work, but the majority of the work can shift to improving the user interface.