Regurgitate: Using GIT For F/LOSS Data Collection

Bart Massey
Computer Science Department
Portland State University
P.O. Box 751 M.S. CMPS
Portland, Oregon USA 97207–0751
bart@cs.pdx.edu

Keith Packard
Open Source Technology Center
Intel Corporation
7615 SW 59th Ave.
Portland, Oregon USA 97219–1204
keith.packard@intel.com

ABSTRACT

We have created a new tool, *regurgitate*, for importing CVS repositories into the GIT source code management system. Important features of GIT include great expressiveness in capturing relationships between revisions and across files as well as extremely high-speed processing. These features make GIT an ideal platform for gathering detailed longitudinal metrics for open source projects. The availability of regurgitate facilitates using GIT as an analysis tool for that majority of open source projects that keep their repositories in CVS. In particular, GIT is fast enough that it is practical to replay the entire development history of a project commit-at-a-time, collecting metrics at each step. We demonstrate this process for a simple metric and a collection of benchmark F/LOSS repositories.

1. THE REPOSITORY MIASMA

The measurement and analysis of software properties of large systems has always been a substantial challenge. The development model used in F/LOSS has exacerbated this challenge. Because F/LOSS software is usually developed by distributed teams, with little formal control over the software engineering process, there is little opportunity to institute metric collection and feed metrics into the process during development.

Further, up until the last five years or so, the principal F/LOSS source code management tools have been RCS [5] and CVS [1]. RCS is a per-file tool with no notion of project-level development at all. While the usual usages of RCS actually retain a good deal of development history, that history is distributed across the various RCS files that comprise the repository. CVS was originally implemented as a collection of shell scripts designed to work as a front-end to RCS for data collection. While it does structure the RCS commit and checkout process somewhat, its principal contribution is merely to provide safe “concurrent” access to the repository by multiple developers. As such, CVS has historically lost a great deal of history at each commit.

Unfortunately, the failure to collect metrics during F/LOSS development limits the methods available to collect historical metric data on F/LOSS projects. In spite of the limited history retained by the tool, analysis of CVS archives generally is the only way forward.

1.1 PROMISE

About a year ago, the authors made an effort to collect and archive some simple metrics on a few F/LOSS projects using the CVSAnalY [3] tool [2] as part of the PROMISE repository work. While some data was collected, superficially analyzed, and made available [4], several problems limited the usefulness of this approach.

First, the data that could be derived from the repository were limited to what CVSAnalY was able to measure. Most metrics collected by the tool have been focused on providing “dashboard-like” snapshots of project progress, rather than the sort of data needed to analyze project engineering, such as McCabe-style code complexity measurements. Second, the data in the repositories was quite “noisy”. Because CVS until recently lacked any notion of *atomic commit*, the granularity of what measurements there were was at the per-file commit level. While the commits were automatically grouped by CVSAnalY, the algorithm used by the tool was not particularly robust in dealing with the many arcane and history-shrouded complexities of the CVS format, nor with the tendency of CVS repositories to be manually edited by developers to work around tool deficiencies.

(None of this commentary should be taken as a critique of CVSAnalY, which the authors of this paper were using well outside its intended scope. In addition, a new version of CVSAnalY has been released more recently; the authors have not yet had an opportunity to evaluate this new version, but suspect that it may ameliorate some of the problems described above.)

Finally, the CVS repositories themselves were missing a lot of data. While there is not much that could have been done about this, it did emphasize the need of analysis to infer as much information as possible given missing data.

The limitations of the PROMISE experiment were manifest. It seemed clear that proper analysis of F/LOSS CVS repositories would require a more direct approach. Due consideration was given, and the project was set aside temporarily.

1.2 GIT

In the past 5 years or so, a number of F/LOSS projects have sought to provide better source code control for distributed development. Of these, GIT is emerging as the likely replacement for CVS in F/LOSS work.

GIT is the result of recent efforts by Linux Torvalds and his associates to provide distributed source code management suitable for Linux kernel development. The requirements for this project are intense. In particular, extremely high tool execution speed and extreme flexibility are essential given the development methodologies used by the kernel team. GIT answers this challenge admirably.

The interesting thing is that the properties that make GIT ideal for kernel development also make it ideal for collecting software metrics. Because GIT can replay the change history of an entire large repository extremely rapidly, the opportunity arises to collect per-commit metrics directly from the source version. Because GIT was designed for a complex development environment, it can store and use almost the full information about each development step, making replays extremely accurate.

However, since most existing F/LOSS repositories are CVS-based it is difficult to see how to use GIT as a tool for analyzing them. While automatic CVS import tools for GIT have existed since early in the project, they have not been particularly effective in preserving state across the import. As a result, valuable development process data is trapped in the CVS ghetto, making the GIT-based approach seem infeasible.

1.3 Regurgitate

Because the Linux kernel developers only began using source code management tools within the last few years, and because the tool used prior to GIT was Bitkeeper, a substantially similar tool, there was little concern about preserving repository history in the kernel. As other F/LOSS projects began to adopt GIT, they found that they had few palatable options for preserving their source code's change record. As mentioned above, import tools were problematic. It was possible to capture some of the project history through CVS replay, but much data was lost in this process also.

When it was decided to move the modular release of the X.org/freedesktop.org X window system repository to GIT, the problems with GIT import of CVS repositories needed to be addressed in earnest. The X repository history has been quite actively used by X developers for some time. In addition, the sheer size of the repository meant that important defects introduced during import were difficult to identify and time-consuming to correct.

One of the authors (Packard) took on the challenge of constructing a tool suitable for high-speed, high-quality import of CVS repository data. The fruit of this effort is *regurgitate*, a CVS-to-GIT repository importer consisting of about 3500 lines of C programming language code. Regurgitate works at high speed (as illustrated in the tables in section 2 below), while preserving almost all of the information that was present in the CVS repository, and very accurately inferring omitted information such as commit history.

1.3.1 CVS structure

CVS stores a separate revision tree for each file in a repository. The revision history is stored as a tree, with the initial revision of the file forming the root and the current states of the file along each branch as the leaves. The most recent revision of the source along the trunk branch is stored in the file. Other revisions are saved only as incremental deltas backward from that revision.

The branch structure of each source file is explicitly stored in the repository making the reconstruction of the history for any single file reasonably straightforward.

However, branches, tags and commits within the separate revision control files do not reflect any global history, but only the history for each file separately. In addition, the CVS command set is largely free of global repository operations. Thus, the global state of the repository is implicit, and CVS commands must be careful to keep the relevant state synchronized across multiple revision control files.

Lacking explicit global state, the tagging, committing and branching operations must modify each revision control file separately, making these operations cumbersome in a large project. In addition, there is no mechanism provided for recording file name or access control changes. Various adhoc mechanisms are used by developers to rename files. Access control is inferred from the access control of the revision control file; the most recent commit to the file sets the access control state for every revision in history.

1.3.2 GIT structure

GIT stores an entire tree of files as a single object in the repository. A "commit" object represents a revision of the project and references a tree of files, various revision control status (author, message, etc) along with the set of previous revisions from which the current revision was derived. Various compression techniques are employed to minimize the storage requirements for this structure.

As the individual objects within the repository point only into the past, GIT stores branch information by referring to the tip of each branch. Merge and join information are extracted from the repository by walking through the revision DAG.

1.3.3 Converting from CVS to GIT

Given the ease with which the complete development history for a single file can be constructed from the CVS revision control file, it seems like constructing the history for the entire project should be straightforward. Starting with the most recent revision of each file in the project along each branch, the history of the entire project is constructed by creating separate project-wide revisions at each changeset.

Changesets are computed from the CVS revision control history by grouping individual file changes by a given committer that occur at approximately the same time and are annotated with the same change message. The most recent versions of the CVS tool, revisions in the CVS history are also marked with an ID which is written to every file committed in a single CVS invocation. Where present, this

Table 1: Benchmark Repositories

Codebase	Date	Files	LOC	Changesets	
				Total	Mainline
libXau	17 Nov 2003	13	1,377	43	33
XTerm	31 Jan 2004	118	79,959	451	377
XServer	1 Feb 2004	8,057	1,588,290	10,251	8,941
Postgres	24 Mar 2006	5,887	1,953,736	24,574	22,293

'commit id' is preferred to the combination of date and commit message. Oddly, the other piece of global information that is preserved by the CVS tools is the commit revision numbers of vendor branches, which are chosen by the user.

Once the first revision along each branch is located, the join point within the parent branch is computed and the revision history can then be replayed to construct the GIT repository. For well-maintained projects with simple revision history, this turns out to be a well defined operation, yielding an obviously correct result. However, because much care must have been taken by the developers during every CVS operation, there are often discrepancies in the CVS revision control files which complicate this operation and lead to ambiguities.

A common problem found in a CVS repository is the presence of ambiguous branch origins, the situation where different files branch at different times. This happens most often when files are added to the repository after branches are created. In CVS, files are always added to the trunk branch; for them to be present in another branch, they must be separately placed on that branch. This operation creates an apparent origin for the branch when the new file is added to the branch. The "real" branch origin is assumed to be the earliest origin found in any of the files in the CVS repository.

Another difficulty is dealing with "vendor branches" within the CVS repository. These branches hold revisions of the source code as managed by an external entity and are often used to bootstrap a new CVS repository from an existing project. The difficulty here lies in what happens when a new revision is created along a vendor branch. By default, CVS configures any modified file so that the latest version along the vendor branch appears to belong to both that vendor branch and the trunk branch for the file. This muddies the revision branch structure within the CVS file. Merging these two CVS branches into a single GIT branch eliminates the ambiguity at the cost of a slight loss of information.

With these (and a few other minor) ambiguities resolved, the CVS repository can be converted to a GIT repository with a minimal loss of data.

2. EXPERIMENTS

Given the situation described to this point, the hypothesis that regurgitate and GIT would provide a good platform for collecting F/LOSS metric data seems like a good one. In order to test this hypothesis, simple experiments were conducted on a number of open source repositories.

The platform for the experimental work reported in this section was Packard's 1.3GHz Pentium M laptop, with 1GB of RAM and a 100GB 5400 RPM drive. It was felt that this somewhat underpowered platform would provide a fairer basis for evaluation than some high-end server machine; operations that executed quickly on this laptop should execute quickly in any modern environment. Qualitatively, the machine seemed to be CPU-bound rather than I/O-bound for most of the data collection reported here.

The repositories used in the analysis are shown in table 1. The X Window System repositories were taken from the XFree86 project, which afforded long-running CVS archives with a complex structure. They consisted of a tiny, rarely changing baseline project (libXau); a medium-sized, somewhat fluid application (XTerm); and a large systems project (XServer). Because the authors have been involved in X development for many years, confidence that these repositories were being handled correctly was high. The fourth F/LOSS project, the Postgres database, was chosen because it is somewhat larger and with a somewhat longer and more complex history than even the XFree86 X server—in fact, Postgres is something of a worst case in terms of CVS repository size and complexity. Also, Postgres has had almost no development overlap with X, and thus represents something of an independent confirmation of the results obtained on the X repositories. The LOC figures listed represent all lines, including blanks and comments, as reported by the POSIX *wc* command. All of the repositories contain largely C code, and thus the measurement largely reflect C code size and complexity.

The experimental procedure was as follows. First, each CVS repository was imported into GIT using regurgitate. Second, the GIT repository was *packed* into a more efficient representation. This can be an expensive operation, but it is essentially a one-time cost; subsequent re-packing of new GIT commits is rapid, and packing the repository dramatically improves GIT performance. Third, the entire development history of each repository (along the master development branch) was replayed, while recording some simple metrics across each file in the repository. Finally, the development history was replayed again, but with metrics computed incrementally by re-measuring only changed files at each step. This last step was necessary to achieve adequate performance on the larger repositories, and did not add substantially to the difficulty of measurement.

Performance of the regurgitate tool is shown in table 2. It can be seen that the performance on even the large repositories would be adequate for most experimental uses. It should be possible to reduce these times from hours to minutes for subsequent repository imports by making regurgitate do incremental GIT repository update, as suggested in section 3. However, the performance shown is adequate for most soft-

Table 2: Regurgitate Performance

Codebase	Import
libXau	8.5s
XTerm	2:45s
XServer	90:10s
Postgres	96:38s

ware metric studies.

As a demonstration metric, total LOC (as measured above) and statements (as measured by semicolon count) as well as total number of functions (as measured using the ETAGS utility, a tool designed to index functions for the EMACS editor) were recorded. This enabled the computation of average lines per function and statements per function at each development step. While this is perhaps not the most exciting metric possible, it does demonstrate the capacity to easily measure non-trivial properties using existing simple tools.

The measurement performance for the sample repositories is shown in table 3. Replay alone is 2–3 times faster than replay with measurement; measurement speed is the limiting factor. The measurements scripts used here are quite inefficient; carefully-crafted custom measurement tools would presumably provide some speedups.

Note that the plan of measuring the entire repository on every development step is actually faster on the smaller repositories (because of inconsequential tool differences), but becomes infeasible for large repositories. The differential analysis solves that problem fairly handily. As noted above, Postgres is something of a worst case repository; even then, differential measurement completes in a feasible amount of time.

Finally, figure 1 shows estimated mean code lines per function for each project as a function of time. [The data for Postgres is currently incomplete; this will be corrected before the final revision.] There are some interesting artifacts here; nonetheless these results should be taken primarily as evidence of success of the measurement method. Perhaps the most glaring thing to explain in the data is the sudden sharp drop and then recovery in the XTerm data. It turns out that this is due to a bunch of "functions" (actually data mis-identified by ETAGS as functions) being turned into macros by a revision, and that revision being later backed out. Thus, the anomaly in the data is reflective of a real software development issue.

While the reported measurements are exploratory, they do demonstrate the power of the approach. Given current infrastructure, designing and running measurements to answer specific questions should now be straightforward.

3. CONCLUSIONS AND FUTURE WORK

The work reported here shows that the strategy of importing F/LOSS CVS repositories wholesale into GIT using regurgitate and then making software measurements by replay of the development history is feasible and powerful. As more F/LOSS projects move to GIT using regurgitate, this approach also becomes a feasible way to do incremental measurement.

A number of improvements are possible to the existing system. It is likely that significant speedups can be obtained in various places, as noted in previous sections. Regurgitate should soon be modified to import incremental changes to a CVS repository into GIT; this will allow incremental measurement using the approach described here even for F/LOSS projects still stuck in the CVS ghetto. The measurements reported here are for experimental purposes only; someone should derive some interesting results using the approach to demonstrate its usefulness.

Acknowledgements

CVSAnalY proved a nice introduction to the problem domain. The GIT team, especially Carl Worth, provided valuable help in understanding how to use GIT in this way. The 2005 PROMISE Workshop provided an ideal venue for the first stage of this work.

Availability

GIT is freely available in source form under the GPL. See `http://git.or.cz` for more information. Regurgitate is freely available in source form under the GPL. Its GIT repository is at `git://git.freedesktop.org/~keithp/parsecvs` (the name change to "regurgitate" is not yet complete). The measurement scripts used in this work are available at `git://svcs.cs.pdx.edu/storage/measurement`. The open source repositories measured here are available at `http://svcs.cs.pdx.edu/benchmark-repos/`.

4. REFERENCES

[1] K. Fogel and M. Bar. *Open Source Development with CVS.* Coriolis, 2001.

[2] B. Massey. Longitudinal analysis of long–timescale open source repository data. In *Proc. 27th International Conference on Software Engineering (ICSE) Workshop on Predictor Models in Software Engineering (PROMISE 2005)*, St. Louis, MO, May 2005. URL `http://www.cs.pdx.edu/~bart/papers/promise-data.pdf`.

[3] G. Robles, S. Koch, and J. González-Barahona. Remote analysis and measurement of Libre software systems by means of the CVSanalY tool. In *ICSE 2004—Proceedings of the Second International Workshop on Remote Analysis and Measurement of Software Systems (RAMSS '04)*, pages 51–55, Edinburgh, Scotland, 2004.

[4] J. Sayyad Shirabad and T. Menzies. The PROMISE repository of software engineering databases. School of Information Technology and Engineering, University of Ottawa, Canada, 2005.

[5] W. F. Tichy. RCS – A system for version control. *Software – Practice and Experience*, 15(7):637–654, July 1985.

Table 3: Measurement Performance

		Measurement		Rate	
Codebase	Replay	Full	Diff	Full	Diff
libXau	1.3s	4.1s	5.3s	8.0 rev/s	6.2 rev/s
XTerm	26.5s	1:34s	2:03s	4.0 rev/s	3.1 rev/s
XServer	46:55s	> 4 hours	87:36s	-	1.7 rev/s
Postgres	1:27m	> 8 hours	2:15m	-	2.8 rev/s

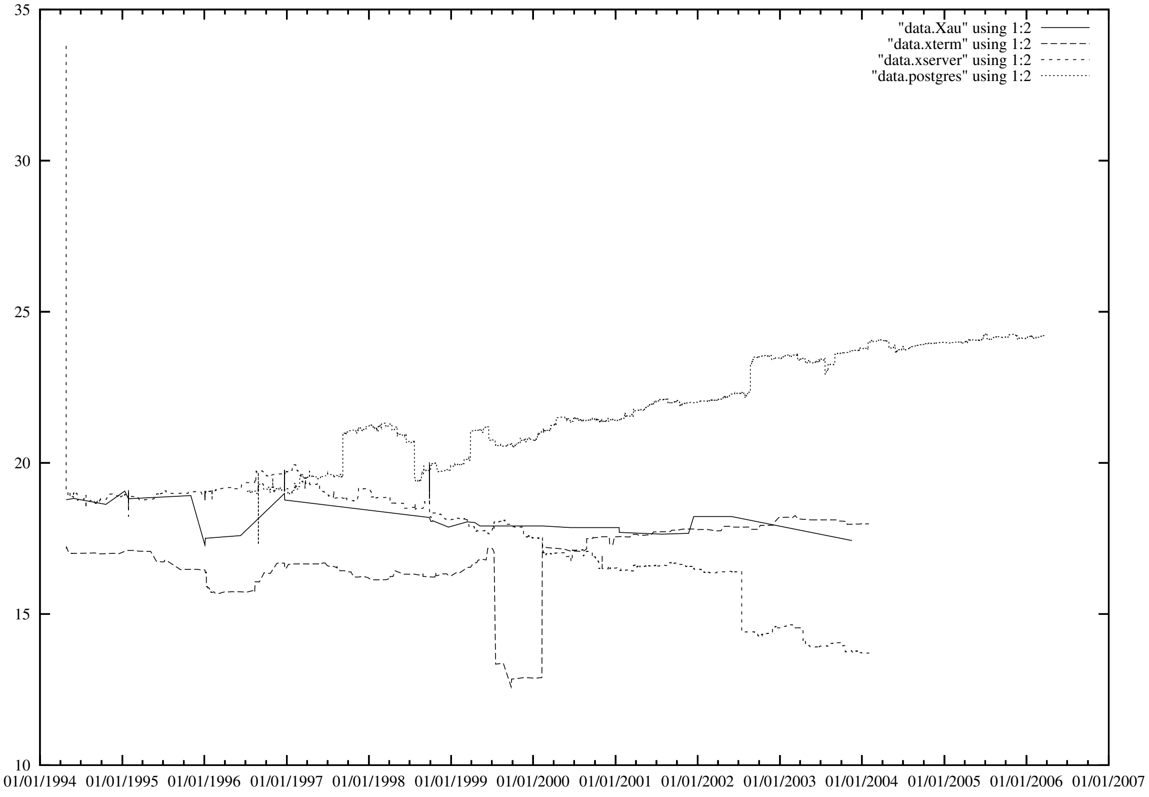

Figure 1: Mean lines per function