Amassing and indexing a large sample of version control systems: towards the census of public source code history

Audris Mockus
Avaya Labs Research
233 Mt Airy Rd, Basking Ridge, NJ 07901
audris@avaya.com

Abstract

The source code and its history represent the output and process of software development activities and are an invaluable resource for study and improvement of software development practice. While individual projects and groups of projects have been extensively analyzed, some fundamental questions, such as the spread of innovation or genealogy of the source code, can be answered only by considering the entire universe of publicly available source code[1] *and its history. We describe methods we developed over the last six years to gather, index, and update an approximation of such a universal repository for publicly accessible version control systems and for the source code inside a large corporation. While challenging, the task is achievable with limited resources. The bottlenecks in network bandwidth, processing, and disk access can be dealt with using inherent parallelism of the tasks and suitable tradeoffs between the amount of storage and computations, but a completely automated discovery of public version control systems may require enticing participation of the sampled projects. Such universal repository would allow studies of global properties and origins of the source code that are not possible through other means.*

1. Introduction

The key output of software development activities is the source code. Therefore, software development is mirrored in the way the source code is created and modified. Given the source code's prominence, it and its evolution have been extensively studied for individual projects or even groups of projects. We would like to be able to apply these and other techniques in a setting approximating the entire universe of all publicly accessible source code in a historic setting. There are two primary advantages of such approximation. First, global properties of the source code, such as reuse patterns, would be difficult to study on a smaller sample of the source code because many instances of reuse may be outside the boundary of smaller samples. Second, many analyses and tools developed for individual projects with a long version history would have insufficient data for projects with an incomplete history of the source code. Therefore, we propose to amass and index a very large sample of source code version histories to support global and historic analysis of public source code. A prerequisite for such analysis involves finding source code repositories, retrieving them, and indexing the source code by content. A desired result would include benefit to the public and the software projects involved.

Here we present lessons learned during six years of experiments in amassing a large sample of public and corporate VCSs and propose several approaches on how such infrastructure could be collected and used, including what we have already implemented.

We start by discussing related work in Section 2 and outline our approach in Section 3. Methods used to discover publicly accessible version control repositories are presented in Section 4. Section 5.1 considers how to retrieve these repositories for further processing and Section 5.2 discusses problems we encountered indexing the source code by its content. Section 5.3 considers how such repository was updated. Issues of robustness considered in in Section 5.4 arise due to the diversity of data sources, large volume of data, unreliability of network connections, and extended processing times. Section 5.5 looks at the constraints imposed by ethical considerations to minimize the impact and to maximize the value for the sampled projects. Different stages of the analysis require different types of system resources, somewhat complicating the overall solution. Section 6 describes infrastructure we used to collect and process the data and propose alternative architectures that are likely to improve performance. In Section 7 we de-

[1] We pragmatically and for brevity refer to all publicly available source code, though a more formal and commonly used, though much longer, term is "free, libre, and open-source software."

scribe in more detail several applications of this data, and conclude in Section 8.

2. Related work

In addition to well-known forges such as SourceForge, Savannah, and GoogleCode that provide valuable public service offering resources for projects to host their code and problem tracking systems, there are several research oriented projects that are collecting a variety of metadata associated with public projects. For example, FLOSSMole [8] collects the list of projects and their metadata from SourceForge, FreshMeat, Free Software Foundation (FSF), RubyForge, and ObjectWeb. SourceForge.net Research Data Archive [15] collects problem reporting systems. An attempt to quantify the extent of reuse in open source is described by Mockus [11], though it had only five million file-versions in the repository. There have been other attempts to gather source code for large project samples, for example, Hahsler [7], but they do not appear to be representative or of reasonable magnitude (Hahsler [7] analyzed 12K SourceForge projects). Koch [10] reports on an EU funded project that collects software metrics for open source projects and aims eventually to include up to 5000 projects. These numbers represent more than an order of magnitude fewer projects than an approximation of a complete sample would have.

It is worth noting that use of open source code in commercial projects imposes a number of responsibilities on the user and detection of such use represents business opportunities. Several companies, for example, Palamida (www.palamida.com) and Black Duck (www.blackducksoftware.com), have a business model based on providing tools and consulting in this area and, therefore, must collect and index public source code. However, their objectives are primarily focused on compliance with licensing requirements and methods are not transparent because, presumably, they represent their competitive advantages. However, Black Duck does provide occasional public releases prominently indicating the number of web sites and the number of projects they consider. It, however, appears that the database they construct, apart from being not accessible, is not suitable for historic analysis afforded by version control systems. A business model based on advertising is embodied by Google Code Search tool (codesearch.google.com) that allows search of publicly available source code by language, package name, file name, and types of license. The service helps finding reusable code and, presumably, will be supported by advertising revenue as are other types of search services provided by the company. The size of the repository and the ranking algorithms are not published. We should note that Google Code Search tool (www.google.com/codesearch) does not search for version control systems and it is not GoogleCode forge (code.google.com) discussed elsewhere in the paper.

3. Outline

First, we need to discover public source code and associated VCSs. Given the large number of projects with public source code and typically obscure way access to project's VCS is published, this step presents a significant challenge. We focus on investigating independent sources of information and discovering sites (forges) with large collections of version control systems.

Second, we need to collect (and keep updating) data from most or all of publicly accessible version control systems. Given the large number of projects ($>$100K in SourceForge and $>$70K in GoogleCode), a variety of repositories (many large projects are not on SourceForge or GoogleCode), and the amount of data (CVS from SourceForge alone occupies approximately 1TB) it represents another serious challenge. Possibly because of that, the existing efforts focus on metadata (FLOSSMole [8]) or problem reporting systems (SourceForge.net Research Data Archive [15]), yet version control systems represent the primary end result of software development activities.

After the version control data is collected it needs to be analyzed further to make it suitable for measurement and analysis. This implies constructing a database for all versions of all source code files. For example, a single iteration over all the files in the retrieved clones of these hundreds of thousands of VCSs would take months. Many analyses may require multiple iterations, for example, finding relationships among this vast collection of file/versions to find instances of reuse. To facilitate similar global analyses we construct a database indexed by the content of file/versions. Such database could also be used to extract source code metrics, a simplified representations of the file content, including removal of multiple spaces, obtaining Abstract Syntax Trees (ASTs), and other summaries of the content. The construction of such database is discussed in Section 6.2.

We rely on the software data analysis techniques described in, for example, [12] that outline approaches needed to obtain valid results from software repository data. We used a hierarchical methodology [13] with more labor/computer resource intensive techniques applied on smaller samples in order to validate less precise and less labor/resource intensive techniques that can be applied on a larger sample. Another approach that we use is triangulation of the methods whereby the results obtained using one method are compared with the results obtained using a qualitatively different method or source of data. Finally, given the open-ended nature of future investigations, we chose to preserve raw data and scripts for each extract. The data was classified according to the level of process-

ing applied. At Level 0 was the raw data obtained from the web sites, relevant URL's, retrieved documents, and other metadata such as version control logs, and clones of the version control systems. Level 1 data included minimally processed/optimized data, such as the tables of file changes (using VCS-specific schema) and content-based indexes for individual projects or parts of projects. At Level 2, we had a complete content-based index presently containing 77,819,492 distinct contents of source code for 207,904,557 file/versions. Results of further analysis, would produce data at Level 3 and above.

4. Discovering forges

Our initial search strategy was based on investigating independent sources of information to discover sites with large collections of version control systems. We started from well-known project portals (forges) such as SourceForge, GoogleCode, Savannah, and Tigris. Some of the repositories supported multiple version control systems. In particular, Savannah supports CVS, Subversion, Git, Mercurial, and Bazaar, though most projects use CVS.

Many of the large and and well-known projects including Linux kernel, FreeBSD, NetBSD, OpenBSD, OpenSolaris, Gnome, KDE, Eclipse, RubyForge, OpenSolaris, NetBeans, OpenJDK, and Mozilla also had mini-forges involving projects related to the main project. Some, for example, Mysql, Perl, Wine, Postgres, and GCC, did not have an associated mini-forge.

An approach to capture the most widely used projects was focused on obtaining source code and project information from popular distributions. In particular, Gentoo, Debian, Slackware, OpenSuse, and RedHat distributions and package repositories such as rpmforge, provided a list of popular packages.

The fourth source of forges we discovered by investigating third generation version control systems such as Git. They had forges associated with that particular version control systems at, for example, repo.or.cz, github.com, gitorious.org, and git.debian.org. The list of URLs pointing to project-specific VCS could be constructed using project IDs obtained from the index page.

We looked at published literature for a survey of projects and found, in, for example, by Koch [10], other VCS repositories including wireshark, mono, gnustep, python, plt, samba, gnucash, rastasoft, lyx, ompi, scipy, svn, ACE, fpc, and freeciv. While most of the projects listed in the published literature were on the forges we have already discovered, there were individual projects and forges that have not been encountered before, for example, common-lisp.net, cvs.kaffe.org, cvs.xemacs.org, freedesktop.org, and sourceware.

In addition to forges that host VCSs, there are directories of open source projects that list home page URLs and other information about the projects. RawMeat and FSF are two prominent examples of such directories. While they do not host VCSs or even provide URLs to a VCS, they do provide pointers to source code snapshots in tarballs and project home pages. We ran scripts to obtain source code and project pages and investigated frequent URL patterns to discover additional forges that have not been obtained via previous methods.

Finally, to verify completeness of forge selection, we have searched for common filenames (main.c, config.h, etc.) using Google Code Search tool and recorded the URLs for the source code tarballs.

While we did the forge search mostly manually, it is desirable to automate it using a search engine in the future. CVS, Subversion, Git, Mercurial, and Bazaar all have a specific pattern for the URLs pointing to a project repository (examples are presented in Table 1). A spider utilizing a search engine, could grab these URLs from projects' pages. Then mirroring of the VCS or a direct extraction of each version of the source code could be initiated. The list of home pages from open source directories such as FSF or FLOSSMole could provide a focused set of web pages for the search.

In the future, to get a more complete list of public version control systems it may be necessary to recruit the participation of the individual projects by providing the URL where the version control data could be collected. To increase the willingness of individual projects to participate, some compelling benefit to the project needs to be provided. The results of the proposed analysis of global code properties may serve such a purpose. For example, providing an indication of where the code may have come from and what security issues may have been discovered in the code with similar origins.

For comparison, the discovery of version control systems in a corporate environment is in some ways simpler. For one, often there are central repositories such as ClearCase to enable developers from multiple locations to access the source code. Second, the VCS licensing requirements require lists of developers authorized to use a particular tool. A request to the IT department to produce such a list can narrow down the number of individuals to contact. Most projects, as in open source, have Wiki pages describing access to the source code, and a search (if available) over internal web pages would lead to the discovery of VCS's URLs. Finally, a simple old-fashioned human trace from products, to development groups, to VCS repositories can yield excellent results. There are some difficulties as well. Some proprietary VCSs may require large fees to get an additional user license. Often that makes little business sense if used simply for data collection. Some projects are using tools

	URL pattern	Clone	Log	Content
CVS	d:pserver:user@cvs.repo.org:/	rsync	cvs log	rcs -pREV FILE
Subversion	{svn,http}://PRJ.repo.org/	svm sync URL	svn log -v URL	svn cat -rREV URL/FILE@REV
Git	git://git.repo.org/	git clone URL PRJ	git log OPTIONS	git show REV:FILE
Mercurial	hg://hg.repo.org/	hg clone URL	hg log -v	hg cat -rREV FILE
Bazaar	http://bzr.repo.org/	bzr branch URL	bzr log −−long	bzr cat -rREV FILE

Table 1. The VCS-specific commands and URL patterns.

that are not designed for intranet access, making automatic data collection more difficult or impossible.

After the sample of projects is selected, it is necessary to retrieve their version control repositories or code snapshots for further processing. Not all forges provide a list of their projects, and even fewer of them provide a list of their version control systems. In the next section we discuss approaches we used to find that information for the set of forges discovered in the first step.

4.1. Finding project version control systems within forges

To get the list of projects for SourceForge we used FLOSSMole project [8] that collects the list of SourceForge projects and their metadata. We use project ID, because it was used to specify the VCS URL within the SourceForge repository.

Projects utilizing Git VCS tended to list all forge's subprojects in a single file. Many of the Subversion and other repositories also have a single file listing the projects in the forge. Retrieving such file and extracting project IDs is usually sufficient to construct a URL for each project's VCS.

A notable exception was GoogleCode, where only a search interface was available. To discover the list of projects we wrote a spider that starts search of GoogleCode using all singleton alphanumeric characters and classes of projects found on the front page of GoogleCode. Each search results in a list of web pages through which the spider iterates and retrieves all project identifiers and category labels listed therein. These labels are then used as additional search terms and new project IDs and labels are collected iteratively until no new labels are generated. We needed less than ten iterations the three times we ran the search.

5. Retrieval and Indexing

While the latest VCSs (Git, Mercurial, and Bazaar) all have clone functionality to create a replica of the repository, Subversion and CVS were more problematic. CVS does not have a mirroring functionality, but many forges, for example, SourceForge and Savannah had rsync access that enabled us to copy the files from the CVS repository. Subversion does not have mirroring functionality and the rsync or other direct file copy is not feasible because the live repository might change during the copy operation leading to an inconsistent state of the mirror. To clone Subversion repositories we used svm package. Most Subversion repositories allowed mirroring using this approach. However, some repositories did not allow mirroring, in particular, GoogleCode and Apache. We, therefore, had to write scripts to extract the content of each version of each file in the repository using a single process (we used three parallel threads for GoogleCode). Unfortunately, due to bandwidth limitations, Apache project had rules to prevent even that low level of access, therefore we did not include Apache in our sample at this time [2].

All version control systems in the sample (CVS, Subversion, Git, Mercurial, and Bazaar) provide APIs to list and extract all versions of the source code. We have first extracted the list of versions and associated metadata using "cvs log" and analogous commands (see Table 1 for other types of VCSs) and processed the output into a table format with each row including file, version, date, author, and the comment for that change.

It is reasonable to assume that other VCSs will be introduced and used in the future, but it is hard to imagine that such essential features as cloning and listing of the files and changes would disappear. While CVS does not have cloning features (it is easy to simply copy the repository files), Subversion has some rudimentary mirroring features. Third generation systems all support cloning of the repositories. Therefore, to support a new VCS would simply require adding its API for cloning and listing of the repository and extraction of critical fields such as file, version, date, and author.

Table 2 lists summaries of several forges ordered by the number of file/versions. Data for only one type of VCS is presented for each forge. The number of projects in a forge was based on the definitions used by the forge. For example, it would not be unreasonable to assume that KDE contains a very large number of projects, but because Subversion access was available for the entire repository, we consider it as a single project. The number of files was counted by the number of unique pathnames in the entire version history.

[2] The effort to make Apache VCS data public is ongoing, and we expect this situation to change in the near future

Forge	Type	VCSs	Files	File/Versions	Unique File/Versions	Disk Space
git.kernel.org	Git	595	12,974,502	97,585,997	856,920	205GB
SourceForge	CVS	121,389	26,095,113	81,239,047	39,550,624	820GB
netbeans	Mercurial	57	185,039	23,847,028	492,675	69GB
github.com	Git	29,015	5,694,237	18,986,007	7,076,410	154GB
repo.or.cz	Git	1,867	2519529	11,068,696	5,115,195	43GB
Kde	Subversion	1	2,645,452	10,162,006	527,7284	50GB
code.google	Subversion	42,571	5,675,249	14,368,836	8,584,294	remote
gitorious.org	Git	1,098	1,229,185	4,896,943	1,749,991	20GB
Gcc	Subversion	1	3,758,856	4,803,695	395,854	14GB
Debian	Git	1662	1,058,120	4,741,273	1,863,796	19GB
gnome.org	Subversion	566	1,284,074	3,981,198	1,412,849	1GB
Savannah	CVS	2,946	852,462	3,623,674	2,345,445	25GB
forge.objectweb.org	Subversion	93	1,778,598	2,287,258	528,938	17GB
Eclipse	CVS	9	729,383	2,127,009	575,017	11GB
SourceWare	CVS	65	213,676	1,459,220	761,963	10GB
OpenSolaris	Mercurial	98	77,469	1,108,338	91,070	9.7GB
rubyforge.org	Subversion	3,825	456,067	807,421	256,425	4.9GB
Freedesktop	CVS	75	139,225	784,021	375,935	4GB
OpenJDK	Mercurial	392	32,273	747,861	60,627	15GB
Mysql-Server	Bazaar	1	10,786	523,383	133,132	6GB
FreeBSD	CVS	1	196,988	360,876	75,377	2.5GB
ruby-lang	Subversion	1	163,602	271,032	56,935	0.6GB
Mozilla	Mercurial	14	58,110	210,748	105,667	1.6GB
PostgreSQL	CVS	1	6,967	108,905	105,281	0.5GB
Perl	Git	1	11,539	103,157	42,941	0.2GB
Python	Subversion	1	8,601	89,846	76,454	0.8GB
Large company	Various	>200	3,272,639	12,585,503	4,293,590	remote

Table 2. Several large or notable forges.

For example, a file existing on a Subversion branch and on a trunk would be counted twice, but multiple versions of the same file on the trunk would be counted only once. The disk space indicates how much space the clone of the repository occupied on our servers.

It was somewhat unexpected to observe that 595 repositories of Linux kernel have more file/versions than more than 120K projects on SourceForge. Perhaps, this indicates the amount of effort that goes into such a crucial infrastructure project. For comparison, the largest commercial project we have observed had fewer that 4M file/versions over its twenty-year history. All projects in a large enterprise have approximately 12M file versions, with some projects being more than 20 years old. The ratio of the number of file/versions to unique file/version contents reveals the amount of branching in the project, possibly reflecting the number of independent teams on it. The Linux kernel with the ratio of 114 is followed by NetBeans (48) and Gcc, OpenSolaris, and OpenJDK (12).

5.1. Extraction

We use the file and version information from the metadata obtained as described above to extract all versions of files in a repository. All VCSs support a command analogous to "svn cat -rRevision URL" (see Table 1) to extract the actual content of a version of a file. Currently we do not extract all files, in particular, we exclude binary files because of the focus on the source code. For simplicity, we currently recognize binary files using file extension. To create extension-to-binary map we obtained the list of relevant extensions by calculating the frequency of file extensions in all retrieved VCSs. We then classified several hundred most frequent file extensions accounting for the vast majority of files. Many could be immediately classified, for example "c" as a source code and "gif" as a binary extension. For others, we extracted a sample of files from the VCS and used UNIX *file* command to determine file type. We manually inspected the content of files that could not be identified via *file* command. If more than 80% of files in the sample appeared to be source code, we classified the extension as a source code extension. Table 3 lists 25 most common

filenames and 25 most common filename extensions in the sample. The count represents the number of times a particular filename (with pathname stripped) or an extension (the set of characters between the last period and the end of the filename) occur in the sample of unique files (including the pathname) in the retrieved sample. Top filenames represent project build (Makefile, Kconfig), information (README, index.html, ChangeLog), and version control configuration (for example, .cvsignore, config, module, .gitignore) files. Top extension are probably influenced by the massive number of files in the Linux kernel.

Table 3. The most frequent file names and extensions from the sample of 69,383,897 files.

Filename	Count	Extension	Count
Makefile	830708	h	9928878
Makefile.am	286475	c	9346976
README	219482	*no ext.*	6592172
.cvsignore	212947	java	6098491
Kconfig	206477	html	3033381
index.html	156137	rb	2341827
Makefile.in	137793	php	2036351
config	135928	xml	1712703
modules	125485	gif	1683693
notify	123911	png	1676220
loginfo	123896	txt	1408532
cvswrappers	123872	cpp	1345950
taginfo	123859	po	1097343
rcsinfo	123859	js	1005263
checkoutlist	123859	py	676858
commitinfo	123852	cs	670455
verifymsg	123841	S	658867
editinfo	123660	C	515919
ChangeLog	107294	class	501553
setup.c	77078	cc	474044
__init__.py	71456	hpp	445235
package.html	64453	in	394343
irq.c	63601	svn-base	389480
io.h	60021	jar	364561
.gitignore	56892	css	351519

It is worth noting that extracting all versions of files from a repository is a fairly time consuming operation. While CVS, Git, and Subversion are relatively fast looking from this perspective, Mercurial and Bazaar are not. For example, approximately half a million of file/versions in the Bazaar repository of mysql-server took more than one month to extract using a single processing thread from a cloned repository on a local server. The Linux kernel Git mini-forge with 585 projects and approximately 100M file versions took slightly less time (also on a local server) and, thus, was at least 200 times faster.

5.2. Indexing source code

Even when retrieved to local servers, the access to the source code is not straightforward because of the diversity of VCSs and projects.

One of our primary objectives was to join VCSs into a single Universal VCS by detecting instances of file copy across multiple VCS as, for example done in Chang [2]. This requires detection of identical or similar file content. Given more than 80 million unique contents in our repository, a fast algorithms was needed. Prior experiments showed that text retrieval techniques used in document search were inadequate to identify similarity of source code files [3]. Our approach was to use techniques based on multiple hashes, whereby a neighborhood of the file would be determined by collisions in a subset of these hashes. In other words, all files that hash to the same value would belong to the same neighborhood. In the simplest case, if we construct a content-based hashtable, each content would be associated with all file/versions having it, thus eliminating the need to search for identical content. Andoni [1] shows that creating multiple such hashtables based on a simplified content, for example, each hashtable utilizing only a subset of the content, could be used in combination to severely reduce the search space and approximate a variety of content similarity functions. Based on experiments by Chang [3], in addition to file content we use Abstract Syntax Tree (AST) and Nilsimsa hash [4] extracted from the content as two additional hashtable indexes.

Given that content-based index proved much more challenging to create than other indexes and because it was used as a basis to compute the remaining indexes, we focus on its creation in the remainder of the section. Our initial choice was to use Berkeley DB via Perl package DBFile with the key being the content and the value being the semicolon separated list of file pathnames appended with version numbers corresponding to that content. While adequate for indexing source code in a large corporation, it did not scale up to the sample of public projects we obtained. Iterating over the entire hashtable would have taken at least six months. Upon investigation, it turned out that the speed bottleneck was the disk access, because the iterator over the hashtable does not return contiguous locations on the disk. By increasing the caching parameters of the hashtable we could substantially speed-up the iterator for hashtables up to ten times the size of the memory cache, but given the RAM size of 32GB on our server, that was not sufficient for the table taking up more than two terabytes of disk space. After some experimentation with various options to speed up creation and processing of these hashtables, we ended up with the structure shown in Figure 1.

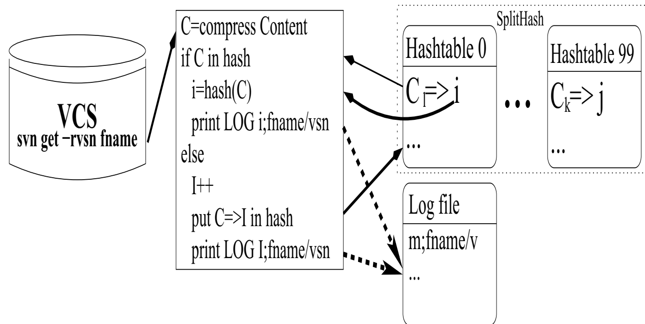

Figure 1. The creation of the content-indexed hashtable.

The content of the file was compressed before being placed in the hashtable as a key. The value for that key was an integer that reflected the order in which the content was inserted. To keep track of filenames and their versions we used a log file that, each time a content was to be inserted into the hashtable, would get a line containing the integer and the corresponding filename/version pair. The integer would be either a new integer (obtained by incrementing the counter) if the content has not been seen before, or the integer value stored in the hashtable corresponding to that content. In the latter case, hashtable did not need to be modified at all. Finally, we used SplitHash module in Perl to pre-hash the compressed content to store it into one of 100 different hashtables. This approach had two advantages. First, we could store different hashtables on different filesystems reducing the bottleneck of disk access while creating the table. Second, we could post-process the 100 resulting hashtables in parallel, increasing the complete iteration over the entire repository up to 100 times.

5.3. Updating

While creating the initial repository was not trivial, updating it exposed additional difficulties. An update started eight months prior to the writing required forge and project discovery as in the original retrieval. The discovery phase revealed a number of new forges from third generation VCSs: Git, Mercurial, and Bazaar. We also discovered more SourceForge and GoogleCode projects. For example, between February and September 2008, the number of GoogleCode projects increased from 50K to 85K. It may represent a stunning growth rate in the number of projects, an improvement in search coverage that allowed capturing a larger portion of the hosted projects via search algorithm that we used, or both. The number of nonempty repositories in GoogleCode increased by a slightly smaller percentage from 30K to 42K. The number of CVS repositories in SourceForge increased to 121K by May, 2008. Some projects from an earlier extract were no longer on SourceForge. At the time of update, Savannah has started providing rsync access to CVS repositories and also supported multiple VCSs for projects to use including Git, Mercurial, Subversion, and Bazaar.

To reduce network bandwidth we wanted to avoid retrieving old files again. Because rsync transfers and VCS cloning functions perform incremental updates, this was not an issue, except for GoogleCode. Because mirroring of GoogleCode Subversion repositories was not possible, we extracted the list of file/versions and compared that list with an earlier retrieval. Only new file/versions were retrieved in the update.

5.4. Robustness

Even though the operations of retrieving and indexing source code are relatively simple, they do take substantial amount of time. For example, just the search for the list of projects in GoogleCode takes a few days using 20Mbps connection. Obtaining the CVS files for SourceForge using rsync takes several weeks. Processing retrieved repositories takes several months even when done in parallel. Over such periods of time network interruptions are common and power outages (despite the use of UPS) are not unusual. Therefore, we avoided performing any operations on the primary copy of version control repositories or of the constructed index, except for replacing it with the updated version once we verified that the operation was successful. A simple file copy of the working version of the index to backup took several hours.

Some operations were fragile. Subversion mirroring process tended to get stuck or terminate apparently normally but before completion. We had to create a supervisor process that would determine if the mirroring locked up or terminated prematurely. The supervisor process would then fix the Subversion database if it was left in an inconsistent state and would restart the mirroring process.

To complete the entire task we had to break it down into smaller chunks and then we could run chunks in parallel and verified the integrity of each chunk before merging them or copying them into the primary location. The nature and size of chunks was determined using a variety of considerations. We broke the retrieval tasks into chunks based on the maximum time we estimated it to run and then classified chunks into groups based on what tasks could be run in parallel. For example, SourceForge, GoogleCode, Savannah, and Linux kernel Git repository cloning could be run at the same time. We restricted Subversion access to GoogleCode and rsync access to SourceForge to no more than three threads simultaneously for each, due to consideration described in Section 5.5. A simple supervisor was written to execute these chunks without violating the constraints and using maximal

specified computing capacity by executing on each server fewer number of processes than the number of processor cores. The remaining processing power was left for other tasks. The number of unused processors depended on the time of day, the day of the week, and the extent to which the server was shared with other activities.

To deal with network problems we kept track of network connection on our side and also kept track of errors in making connections and of zero length retrieved files. When encountering any of these problems while retrieving a chunk of operations we would restart retrieval of the chunk again.

Because extracting content of each version of each file from a version control repository was quite computationally intensive, we created separate hashtables for each project or a smaller repository first, before merging them into the main hashtable. Extracting all versions of all source code files from retrieved repositories would have taken us several months using fourteen processors. As things happen in real life, computer crashes, network problems, and coding errors extended it to a period closer to an entire year.

5.5. Ethics

While we expect a clear public benefit from analyzing a very large sample of source code version history, we need to consider the concerns of the parties that provide the data. When opportunity allowed, we consulted project participants, but it was not feasible to do for all projects given the explicit goal of complete coverage akin to census. Therefore, we chose to limit utilization of the version control systems to make sure it will be under the noise threshold and would not pose any disruptions or overload of project systems. In particular, we had only a single thread retrieving the data from a repository with periodic pauses. For major forges SourceForge and GoogleCode we ran up to three threads simultaneously without pauses expecting that such load would be insignificant for forges containing around 100K projects. We also sought to use less processor and bandwidth intensive operations of rsync, mirroring, and cloning, leaving more computation and bandwidth intensive extraction operations to our servers.

The ability to attract voluntary participation of the projects in the future would alleviate some of these concerns. Unfortunately, the idea of a complete sample is inherently incompatible with voluntary participation, because the latter is likely to bias the sample. Therefore, even with voluntary participation of the projects, the discovery stage can not be completely eliminated.

6. Infrastructure

First we describe the hardware we used and then we consider a basic infrastructure related to unusual storage and computation requirements for the research tasks, and a higher level infrastructure, for example, by constructing Universal Version History, that would enable application of existing software development models and would enable answering a plethora of new research questions. It is important to note that, for example, a cluster of 100 regular workstations would have simplified the the architecture and implementation tremendously, but we did not have it available and had to use available infrastructure that was an order of magnitude less expensive and less powerful.

6.1. Existing hardware

Our primary server was a Sun Fire V40Z with four Opteron processors and 32GB of RAM running Solaris 10. Except for the 45GB of local disk storage for temporary space and swapping, it had an attached raid array with 0.5TB storage using ZFS with compression, that approximately doubled its effective capacity. Another sever provided four dual-core Xeon processors (eight cores) with 8GB of RAM running CentOS. The remaining three desktop computers had dual-core AMD and Intel processors, 4-8GB of RAM, a local storage of 3TB each, running OpenSUSE and distributed over two sites. The primary storage was provided by Coraid NAS device with 8TB mounted via NFS. The remaining storage was used as a scratch space to increase disk throughput. In particular, we distributed the aforementioned 100 hashtables over four different storage locations mounted over NFS.

All computers, except for the two desktops in the remote location, had dedicated or switched 1Gbps Ethernet connections among themselves and a 20Mbps connection to the remaining two desktops that were used to retrieve VCS from the Internet. The Internet connectivity included a direct symmetric 20Mbps connection shared by the two retrieval servers at the remote site and an OC-3 connection through the firewall for the main site. Because of the delays in establishing a TCP connection through the proxy for the OC-3 connection, most of the data collection that needed frequent establishment of TCP connections was done using the two desktops on a separate site and the resulting data then pulled in to the main site through the proxy. While the primary server and attached storage were quite expensive at the time of purchase around 5 years ago (approximately 25K US$), a solution including the primary and secondary servers and equivalent in computing, RAM, and storage, could be reproduced for less than 10K US$ at the time of writing. The symmetric 20Mbps Internet access connection costs approximately 700US$ per year (Verizon FIOS), suggesting that, while ambitious, the project may be implemented on a tight budget.

6.2. Computational Challenges and Proposed hardware

The existing hardware and network setup evolved over time and were based on the equipment available at the time and primarily used for other tasks. Therefore, we are presenting our considerations of the types of hardware solutions that would be particularly suitable for the task and summarize the above described computational challenges in a more general fashion.

An existing setup described above needs significant improvements to enable continuous updates and hosting to enable public access to the resulting database, possibly through TeraGrid (teragrid.org) or a mixed platform.

We distinguish four tasks that pose distinct demands on the number of CPUs, the amount of RAM and disk storage, and on disk IO throughput and the speed of Internet access.

1. Retrieval and updates of version control repositories require a medium amount of network bandwidth and a medium amount of computing resources, but needs at least ten terabytes of storage. Presently, the update takes less than one month in duration for the largest sites SourceForge, Kde, and, particularly, GoogleCode. Increase in bandwidth may not be sufficient to reduce the interval because some of the retrieval tasks are bounded by the bandwidth of the servers hosting the repositories and the necessity not to interfere with normal operations of these repositories.

2. Extracting the content of each version of every file from all (more than 200K) version control systems in the repository. This step requires substantial computational resources for some systems (e.g., Mercurial), and relatively few resources for the remaining systems. The task is highly parallel and would scale well if disk throughput was sufficient. For cases when the VCS can not be mirrored, it requires a moderate amount of network resources and a limited amount of computing resources (again, due to the need to minimize the impact on the project hosting sites).

3. Construction of the primary content indexes of each version of every file. This step can be speeded up either by having a very large amount of RAM (on the order of 1TB) or by having a very fast write medium (disk cluster) and does not require substantial CPU resources. Presently it requires more than one month on a four processor 32GB RAM sun (Opteron-based) server. The Internet access is not needed and limited computing resources are needed.

4. Iterating over the database to calculate a variety of metrics, find instances of reuse, and other measures of similarity among files requires substantial computations for tasks that are easy to parallelize for a cluster computing environment such as extraction of AST and other summaries, and tasks that may be more difficult to parallelize, such as finding similarities among files. Presently, the duration depends on the complexity of the task, with many tasks that can be easily distributed to multiple servers and, therefore, most complete within two weeks using 14 CPUs on four servers.

To speed-up computations to achieve a complete update cycle durations of less than one month, the following setup may be required. If the main index is stored in 1000 hashtables on separate nodes of a cluster then each hashtable would occupy only 2GB of space and may fit within the RAM of the individual node. The pre-hash process running on one of the nodes would determine which system the particular content string is stored on and would dispatch the request to store or retrieve the pathnames and versions using, for example, MPI framework. This solution assumes that the disk IO is localized to each node. Otherwise, a shared storage bus may be overloaded. Experiments are needed to verify if such solution would deliver on its the promise.

7. Immediate applications

There are numerous software development models that rely on data from project's version history that range from techniques to determine effectiveness of software methods, to ways to estimate developer productivity and predictions of software quality. Unfortunately, many software projects have recently switched version control systems (e.g., from CVS to Subversion) and therefore, the VCS repository may contain only a brief recent history. Furthermore, in cases of code reuse it is rare that a file is copied with its entire history, thus loosing important information about its past. To remedy these problems and to enable more accurate modeling of software projects we intend to construct a Universal Version History (UVH), whereby multiple version control systems are connected for the files that have been copied across repositories either because of the repository change or because of code reuse. Such links in conjunction with the dates of the corresponding versions would allow tracing of code authorship and succession.

In succession, developers take over and use or maintain unfamiliar code. A new developer joining a project, offshoring, or reuse of code from another project are instances of succession. A study by Mockus [13] discovered a number of factors decreasing productivity in a succession. Ability to detect reuse afforded by UVH would improve the detection of succession and would allow comparison of code reuse with other types of code transfer scenarios.

One of the advantages of open source projects is that the source code can be taken and modified by other projects in

case the project is no longer supported or if the required modifications are outside the scope of the original project. The practice of code reuse may bring tremendous savings in effort. If the highly reused source code tends to have better quality and requires less effort to maintain as observed by Mohagheghi [14] and by Devanbu [5], the extent of reuse can serve as a guide of source code's reuse potential and serve as ranking of functionally relevant candidates as proposed by Inoue [9]. Furthermore, if highly reused code and projects have attributes that distinguish them from the low-reuse projects, some of these qualities may guide new projects that strive for their code to be reused more widely. Finally, existing projects may be able to take measures to increase the reuse potential of their code.

Certain features in the VCS would greatly benefit collection of source code data. Cloning functionality in Git, Bazaar, and Mercurial make data collection much more convenient than for CVS and Subversion. Git is the only VCS that has separate change attributes for committer and authors. Keeping track of authorship is particularly important in volunteer communities, and projects would benefit if they would be careful to pay attention to having and carefully entering such data. Having a separate field to enter defect tracking numbers would also benefit the projects that analyze their development process. VCS could provide an API to extract all versions of each file in a single request, because reconstructing older versions may be a recursive operation requiring to reconstruct later versions first. This would be particularly helpful for slower systems such as Mercurial and Bazaar.

The ability to track all parents of a merge in, for example Git, (a typical VCS records only merge-to parent) would benefit the construction of the Universal Version History (UVH) by removing the need to guess or estimate all sources for a merge. Also, Git has the ability to keep track of file renames, including moves to another directory, further simplifying the construction of UVH.

8. Summary

We described our experience amassing and indexing by content a large sample of version control systems. We shared some of the challenges encountered in discovering, retrieving, indexing, and updating this vast collection of repositories. We also proposed several approaches to develop such infrastructure into a continuously updated shared resource and discuss several concrete analyses that may improve the quality and and productivity of the ecosystem of public projects.

References

[1] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Communications of the ACM*, 51(1):117–122, 2008.

[2] H.-F. Chang and A. Mockus. Constructing universal version history. In *ICSE'06 Workshop on Mining Software Repositories*, pages 76–79, Shanghai, China, May 22-23 2006.

[3] H.-F. Chang and A. Mockus. Evaluation of source code copy detection methods on freebsd. In *Mining Software Repositories*. ACM Press, May 10–11 2008.

[4] E. Damiani, S. D. C. di Vimercati, S. Paraboschi, and P. Samarati. An open digest-based technique for spam detection. In *The 2004 International Workshop on Security in Parallel and Distributed Systems*, volume 41 (8), pages 74–83. ACM, 2004.

[5] P. T. Devanbu, S. Karstu, W. L. Melo, and W. Thomas. Analytical and empirical evaluation of software reuse metrics. In *ICSE 1996*, pages 189–199, 1996.

[6] R. Ghosh. Final report. study on the economic impact of open source software on innovation and the competitiveness of the information and communication technologies (ict) sector in the eu. Technical report, UNU-MERIT, NL, 2006.

[7] M. Hahsler and S. Koch. Discussion of a large-scale open source data collection methodology. In *HICSS*, 2005.

[8] J. Howison, M. Conklin, and K. Crowston. Flossmole: A collaborative repository for floss research data and analyses. *International Journal of Information Technology and Web Engineering*, 1(3):17–26, 2006.

[9] K. Inoue, R. Yokomori, H. Fujiwara, T. Yamamoto, M. Matsushita, and S. Kusumoto. Component rank: Relative significance rank for software component search. In *ICSE'03*, pages 14–24, 2003.

[10] S. Koch and S. Dueñas. Free/libre/open source metrics and benchmarking. http://flossmetrics.org/.

[11] A. Mockus. Large-scale code reuse in open source software. In *ICSE'07 Intl. Workshop on Emerging Trends in FLOSS Research and Development*, Minneapolis, Minnesota, May 21 2007.

[12] A. Mockus. Software support tools and experimental work. In V. Basili and et al, editors, *Empirical Software Engineering Issues: Critical Assessments and Future Directions*, volume LNCS 4336, pages 91–99. Springer, 2007.

[13] A. Mockus. Succession: Measuring transfer of code and developer productivity. In *2009 International Conference on Software Engineering*, Vancouver, CA, May 12–22 2009. ACM Press. To appear.

[14] P. Mohagheghi, R. Conradi, O. M. Killi, and H. Schwarz. Empirical study of software reuse vs. defect-density and stability. In *ICSE 2004*, pages 282–292, 2004.

[15] SRDA. Srda: Sourceforge data for academic and scholarly researchers. http://www.nd.edu/ oss/Data/data.html.