REPERTOIRE: A Cross-System Porting Analysis Tool for Forked Software Projects

Baishakhi Ray, Christopher Wiley, Miryung Kim
The University of Texas at Austin
{rayb, thewiley, miryung@ece}.utexas.edu

ABSTRACT

To create a new variant of an existing project, developers often copy an existing codebase and modify it. This process is called software forking. After forking software, developers often port new features or bug fixes from peer projects. Repertoire analyzes repeated work of cross-system porting among forked projects. It takes the version histories as input and identifies ported edits by comparing the content of individual patches. It also shows users the extent of ported edits, where and when the ported edits occurred, which developers ported code from peer projects, and how long it takes for patches to be ported.

1. INTRODUCTION

Software forking occurs when a developer or a group of developers splits off software into separate conceptual entities by copying an existing project. Forking is particularly common in free open source software projects, where differing visions and personality clashes occur without an unifying profit motive. For instance, the forking of FreeBSD, NetBSD, and OpenBSD from 386BSD, the split of XEmacs from GNU Emacs, and the split of LibreOffice from OpenOffice.org are well known forks. Software developed by industry may also be forked to support the needs of multiple customers with different feature requirements.

Forking is often considered to be counter-productive. As multiple peer projects evolve in parallel, developers may need to port similar features or bug fixes from one project to another, incurring duplicate maintenance effort. This paper presents Repertoire, a tool that analyzes the extent and characteristics of cross-system porting. It allows users to analyze the number of lines of code ported from the patches of peer projects, the developers responsible for those ported edits, the time taken to port patches from peer projects, etc. It presents the temporal and spatial characteristics of cross-system porting using various graphical views. It also supports interactive browsing of ported edits. Currently it is fully integrated with the state of the art version control systems such as Git and Mercurial. These analyses are designed to aid managers and architects to make informed decisions about the maintenance of forked software systems.

2. REPERTOIRE FEATURES

Suppose Sheryl is a manager working for the Exemplar corporation, which writes and sells software to enterprise customers. Two years ago, a particularly large customer requested a feature that required extensive modifications to the main product. To accommodate this customer's needs, the company forked the main product and made the necessary custom changes. Since then, a considerable amount of engineering effort has been continually spent to port bug fixes and security patches from the main product. Sheryl is contemplating whether it would be worthwhile to merge the two products back instead duplicating maintenance effort.

Sheryl may need to analyze how the products evolve in parallel and how often cross-system porting occurs. She needs to know where the porting effort is focused on, who are the main developers porting code from peer projects, and how often cross-system porting happens, etc. She needs to know which directories and files mostly consist of ported edits. She may also be interested in knowing how long it takes for bug fixes and security patches to propagate from the main product to the other. These are the questions that Repertoire can help Sheryl to answer. For presentation purposes, we refer to the main project and the forked project as A and B respectively in the following subsections.

Porting Frequency View. Suppose that Sheryl wants to know how often cross-system porting occurs. Given the version histories of A and B, Repertoire visualizes the extent of code ported from one project to another over time. In the Porting Frequency View in Figure 1, the x-axis shows time in months and the y-axis shows the average percentage of ported edits with respect to total edits in each commit. Sheryl may select to see only the edits ported from A to B, only the edits ported from B to A, or both ways. Sheryl may see that 90% of the commits to B are ported from the patches of A, whereas 95% of the commits to A are not ported from B, indicating that most engineers working on B spend their time porting code and little time writing original code. On the other hand, if Sheryl notices that most edits to either system are not ported, then she may conclude that the systems are diverging further apart.

File Distribution View. To figure out where her organization is spending time porting code, Sheryl needs to see which pairs of files share port edits between A and B. Repertoire helps Sheryl by presenting the File Distribution View of the source and target of ported edits. This view is a scatter plot with files from A making up the x-axis and files from B making up the y-axis. A point is plotted at `(x,y)` if there is a ported edit from file `x` to file `y` or vice versa. Users can also grasp the amount of ported edits by inspecting the color of the dot. The darker the color is, the higher the ratio of ported edits to the total lines of code in the file. This allows us to answer which files have the most ported edits and which files have the highest ratio of ported edits. The

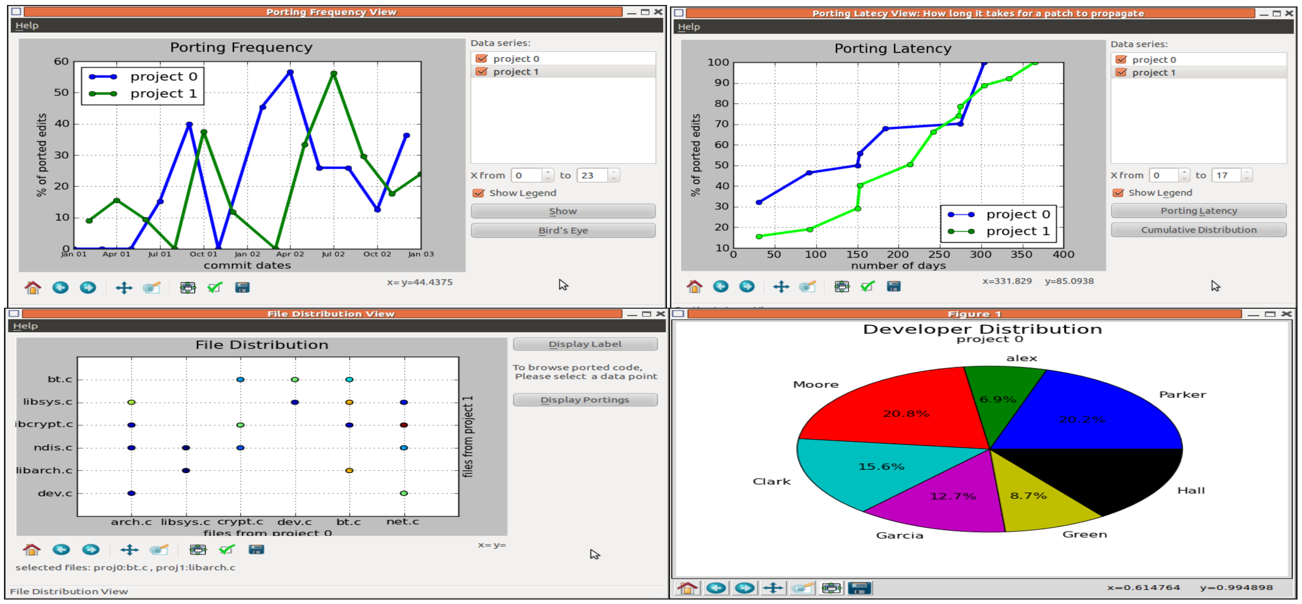


Figure 1: Repertoire analysis of cross-system porting between two forked projects.

File Distribution View in Figure 1 shows an example of this file distribution view. By selecting any point on this view, Sheryl can browse all ported lines between the two files and investigate who ported the corresponding code, the commit dates, etc.

Developer View. To ask her team about the feasibility of merging A and B, Sheryl may need to identify the developers who have a deep understanding of both projects. A reasonable heuristic for finding such developers is to simply identify the developers do a lot of porting work. Repertoire displays a pie chart showing which developers are responsible for what fraction of the total ported lines. See the Developer Distribution View of Figure 1.

Porting Latency View. Repertoire shows a user how long it takes for individual patches to propagate from one system to another system by presenting a cumulative distribution of porting latencies. The Porting Latency View of Figure 1 shows the number of days between when a patch is first committed to one system and when a similar patch is committed to a target system.

3. IMPLEMENTATION AND EVALUATION

Repertoire analyzes *diff*-based program patches of two forked projects to identify the ported edits. It works in two phases. In the first phase, Repertoire uses CCFinderX [2] to identify similar edit contents (clones) in the input patches. In the second phase, Repertoire determines if two identified clones represent similar edit operations by comparing the edit operation types (i.e., addition, deletion, and modification) using an *N-gram* matching algorithm [1].By comparing the commit dates of similarly edited code regions, Repertoire disambiguates the source vs. target of the ported edit. This tool demo paper expands on a tool that we developed to study co-evolution of BSD products and a detailed description of Repertoire is described in our technical report [3].

In 18 years of parallel evolution of the BSD family, on average, FreeBSD ports 13.77% of edited lines from NetBSD and OpenBSD, while 15.52% and 10.74% of edited lines in NetBSD and OpenBSD originate from the other two BSDs respectively. 26.12%, 58.85%, and 44.85% of active developers in FreeBSD, NetBSD and OpenBSD port patches from the other BSDs. The average time taken to port patches from peer projects in FreeBSD, NetBSD and OpenBSD are 734, 725, and 944 days respectively. In all three projects, porting is mostly localized within 20% of the modified files [3].

4. SUMMARY

This paper presents Repertoire that analyzes the extent of cross-system porting among projects forked from a common ancestor. Using Repertoire, managers and engineers can measure the frequency of cross-system porting, learn which developers do how much of the porting work, investigate the trend of cross-system porting work over time, and the spatial distribution of ported edits with respect to the file system structure.

5. REFERENCES

[1] G. W. Adamson and J. Boreham. The use of an association measure based on character structure to identify semantically related pairs of words and document titles. *Information Storage and Retrieval*, 10(7-8):253–260, 1974.

[2] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[3] B. Ray and M. Kim. A case study of cross-system porting in forked projects. In *ESEC/FSE-20: ACM SIGSOFT the 20th International Symposium on the Foundations of Software Engineering*, 2012, to appear.

	Input Types	Example Inputs
	working directory	`/var/tmp`
	CCFinderX path	`/usr/bin/ccfx`
	version control type	Git
Project 1	repository root	`/path/to/myrepo`
	time period	11/2/2010 - 9/31/2011
	version control type	Mercurial
Project 2	repository root	`/path/to/anotherrepo`
	time period	11/2/2010 - 9/31/2011

Table 1: Example inputs to Repertoire.

APPENDIX

The input wizard of Repertoire gathers information about the version histories of forked projects. The analysis wizard of Repertoire then visualizes cross-system porting analysis results between the input projects using several views: Porting Frequency View, Developer View, Porting Latency View, and File Distribution View. Using the inputs specified by the user in the input wizard, Repertoire's back-end extracts individual *diff*-based patches, developers, and commit dates from the version control repositories and compares the content and edit operations of the patches using CCFinderX. Table 1 shows example inputs. After identifying cross-system ported edits, the back-end stores the results in a database, which can then be loaded from GUI visualization components. The internal structure of Repertoire is shown in Figure 2.

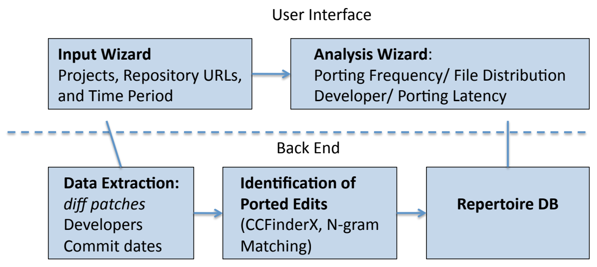

User Interface
Input Wizard
Projects, Repository URLs, and Time Period
Analysis Wizard:
Porting Frequency/ File Distribution
Developer/ Porting Latency
Back End
Data Extraction:
diff patches
Developers
Commit dates
Identification of Ported Edits
(CCFinderX, N-gram Matching)
Repertoire DB

Figure 2: Repertoire internal components.

Repertoire is an open source tool and can be downloaded from `https://github.com/SealLab/RepertoireTool` This section describes the steps required to run Repertoire.

A. INSTALLATION

1. Install required libraries
   - `Python 2.7`
   - `Qt 4.x`: a cross-platform application and UI framework
   - pyuic4: a UI compiler for `Qt` that comes with the `PyQT` package.
2. Run '`make`' from `src/`
3. Run '`make`' from `src/analysis/`
4. Obtain a working copy of CCFinderX for your platform
   - Ensure the execution of CCFinderX by running a command '`ccfx d cpp somefile.cpp`'

B. POPULATING A DATABASE

Repertoire takes as input the repository location and time period of version histories and identifies ported edits among the input projects. It requires a working directory to store intermediate files, a path to an executable CCFinderX, and information about version control repositories. For each repository, the user is asked to specify the type of version control system (e.g. Git or Mercurial), the root URL of the repository, and the time period that the user is interested in. Table 1 shows example inputs. Repertoire checks the validity of inputs and then proceeds to populate a database with the analysis results of ported edits.

1. Run '`src/run_vcs_flow.py`'
   - When an input wizard appears, select "Start a new project"
2. Pick a working directory, e.g. `/var/tmp`.
   - Repertoire creates a sub directory to put its intermediate results.
3. Specify a path to a CCFinder executable.
   - You may optionally pick a minimum token size (CCFinder's input parameter). A minimum token size is the number of lexical token elements that must be similar between two code fragments to be identified as code clones.
4. Select a version control system for each project.
   - Repertoire currently supports Git or Mercurial as target version control systems. Alternatively, a user may provide a directory including pre-extracted *diff*-based patches.
5. Select a URL path for each version control repository.
   - This is the root directory of the repository for Git and Mercurial.
6. Select file extensions for C/C++, headers, and Java files. This step allows users to ignore ported edits in certain types of files.
7. Select a time period for the project. Repertoire then extracts *diff-based* patches for each commit revision within the time period.
8. Confirm analysis of the given data and then wait for analysis to complete.
9. When the analysis is complete, check the output file created by Repertoire in the working directory.
   - There will be a `pickle` file called `rep_db.pickle`, which is a file format for Python object serialization and de-serialization. This is used as an input for the visualization and analysis step.

C. RUNNING REPERTOIRE

1. Run `rep_analysis.py` from `src/analysis`
2. Select the pickle file `rep_db.pickle` produced from the previous step and press `Next`.
3. The GUI provides four analysis views shown in Figure 3: Porting Frequency View, File Distribution View, Developer View, and Porting Latency View (Timing Analysis).

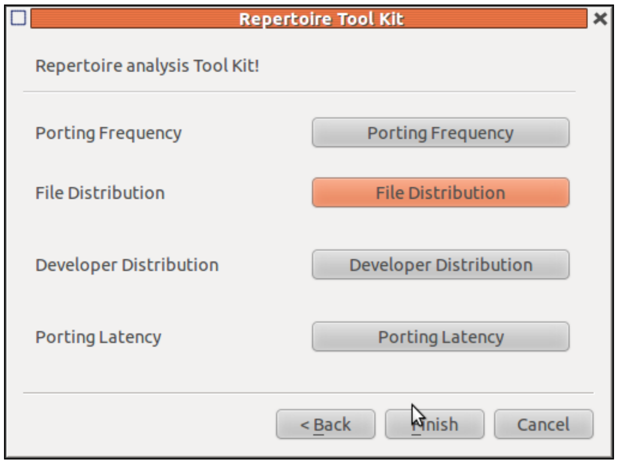


Figure 3: REPERTOIRE **Analysis Menu**

D. PORTING FREQUENCY VIEW

Given the version histories of two projects, this view shows the extent of edits ported from one project to another over the available history. This is represented as a line diagram, where x-axis shows a time line, and the y-axis shows the average percentage of ported edits with respect to total edits in *diff*-based patches. A user may select to see only ported edits from Project A to B, B to A, or both ways at once. Figure 4 shows an example of this porting frequency view. Steps to run this view:

1. Select Porting Frequency in the menu.
2. Select a project: Project 0 orand Project 1
3. Set a time period for analysis.

E. FILE DISTRIBUTION VIEW

This view is a scatter plot where files from Project A is shown along the x-axis and files from Project B is shown along the y-axis. A point is plotted at `(x,y)` if there is an edit ported from file X to file Y or vice versa. The color of the dot indicates a ratio of ported edits to total edits. The darker the color is, the higher density of ported edits. Figure 5 shows an example. Steps to run this view:

1. Select File Distribution in the menu.
2. By default, this view does not show full file names. A user can click Display Label option to see the full file names.
3. When a user click on the point in the diagram, corresponding files names are shown in the bottom.
4. To browse ported code between the selected file pair, press Display Ported Edit
5. A window will show all ported code fragments between the two files, along with developer and commit date information.
6. On selecting any clone from clone list, user can browse the ported edit. Figure 6 shows an example these last two steps.

F. DEVELOPER VIEW

Figure 7 shows an example of developer distribution. The pie chart shows which developers are responsible for what fraction of the total ported lines. The scatter diagram in this figure with developers of project 0 in x-axis and developers of project 1 in y-axis also reflects the interaction pattern of the developers while porting code. Steps to run the developer analysis:

1. Select Developer Distribution in the menu.
2. Shows a scatter plot of developer distribution, i.e., a point is plotted at (x,y) if developers at x port code written by developer at y, and vice versa.
3. We do not show developers names as label initially, as it clutters the display. User can see labels however, if Display Label is pressed.
4. If any point on the diagram is pressed, corresponding developer names can be seen at the bottom.
5. To see developer's distribution in a particular project in the form of pie chart, please select project 0 and/or project 1 from right hand window. Then press Display Developer Porting Statistics"

G. PORTING LATENCY VIEW

This analysis shows how long it takes for a patch to be propagated to the other project on average. Figure 8 an example of this view. Steps to run this analysis:

1. Select Porting Latency in the menu.
2. Select a project (e.g. Project A or B), and then press Porting Latency button.
3. A cumulative distribution of the time taken to port edits from the source to target projects is shown when pressing Cumulative Distribution.
4. A user may limit the time period to a specific time period for an in-depth investigation.

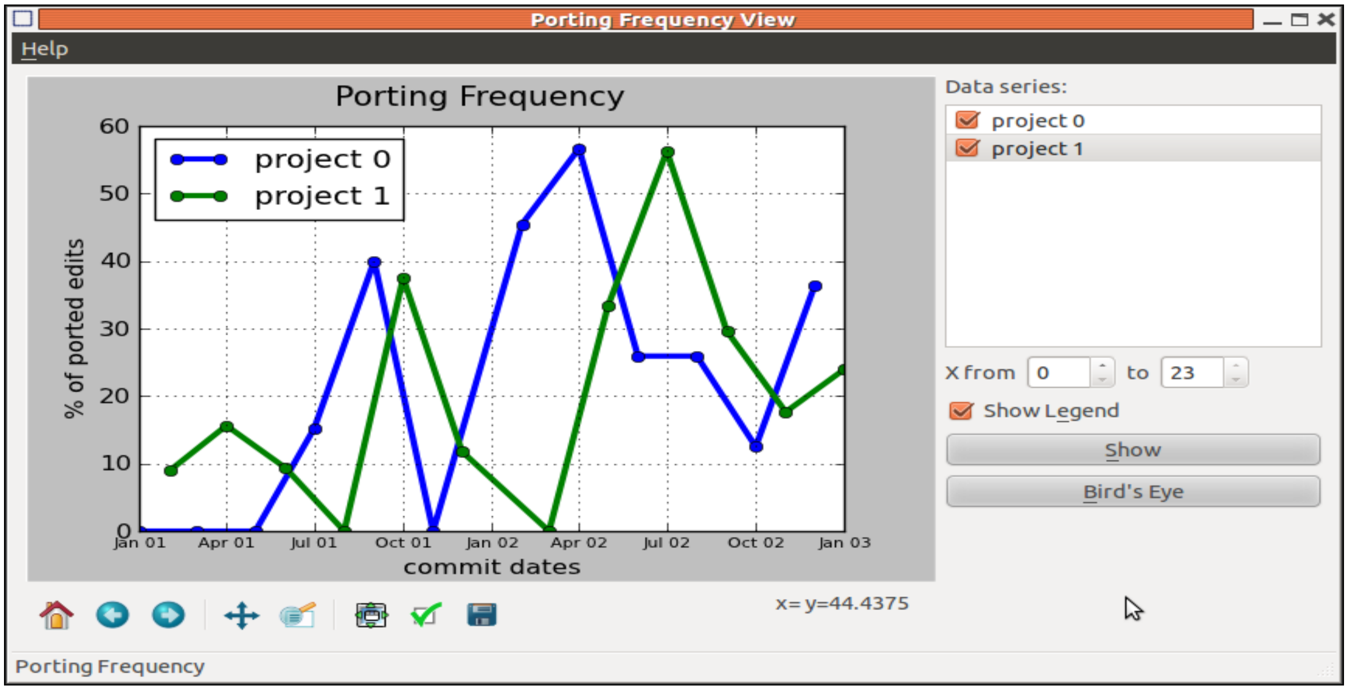
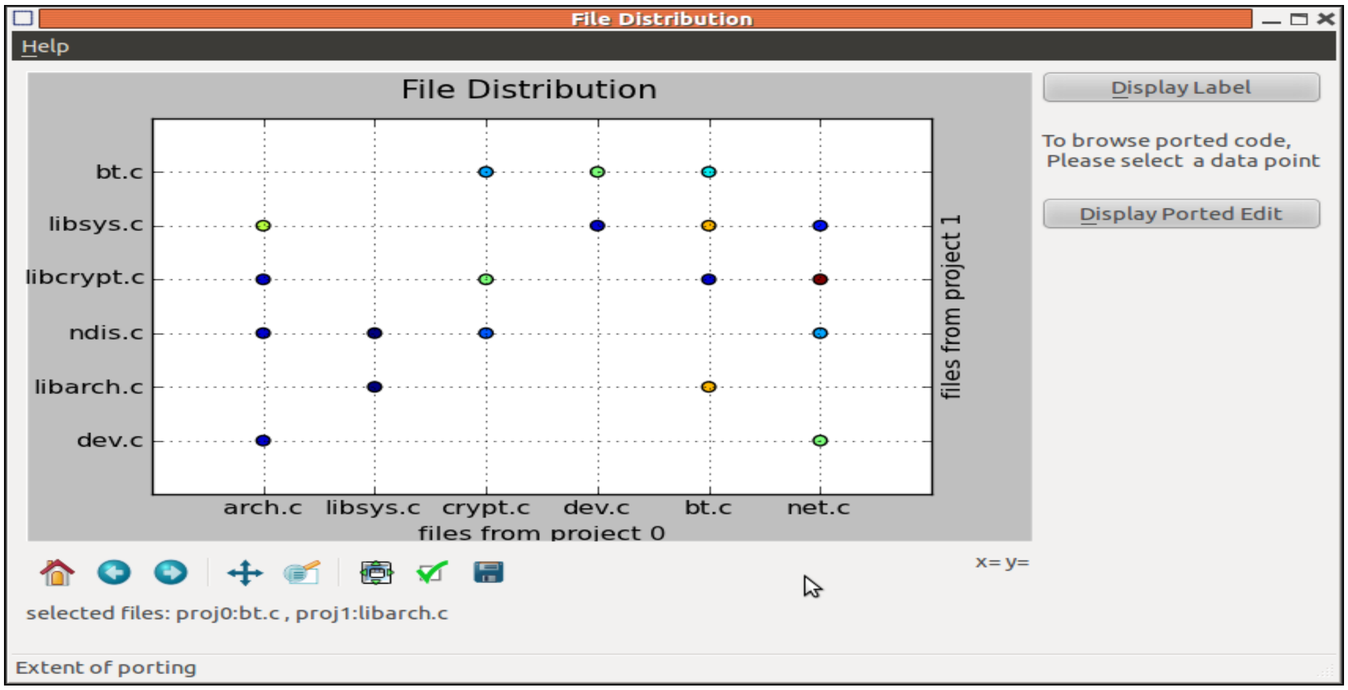
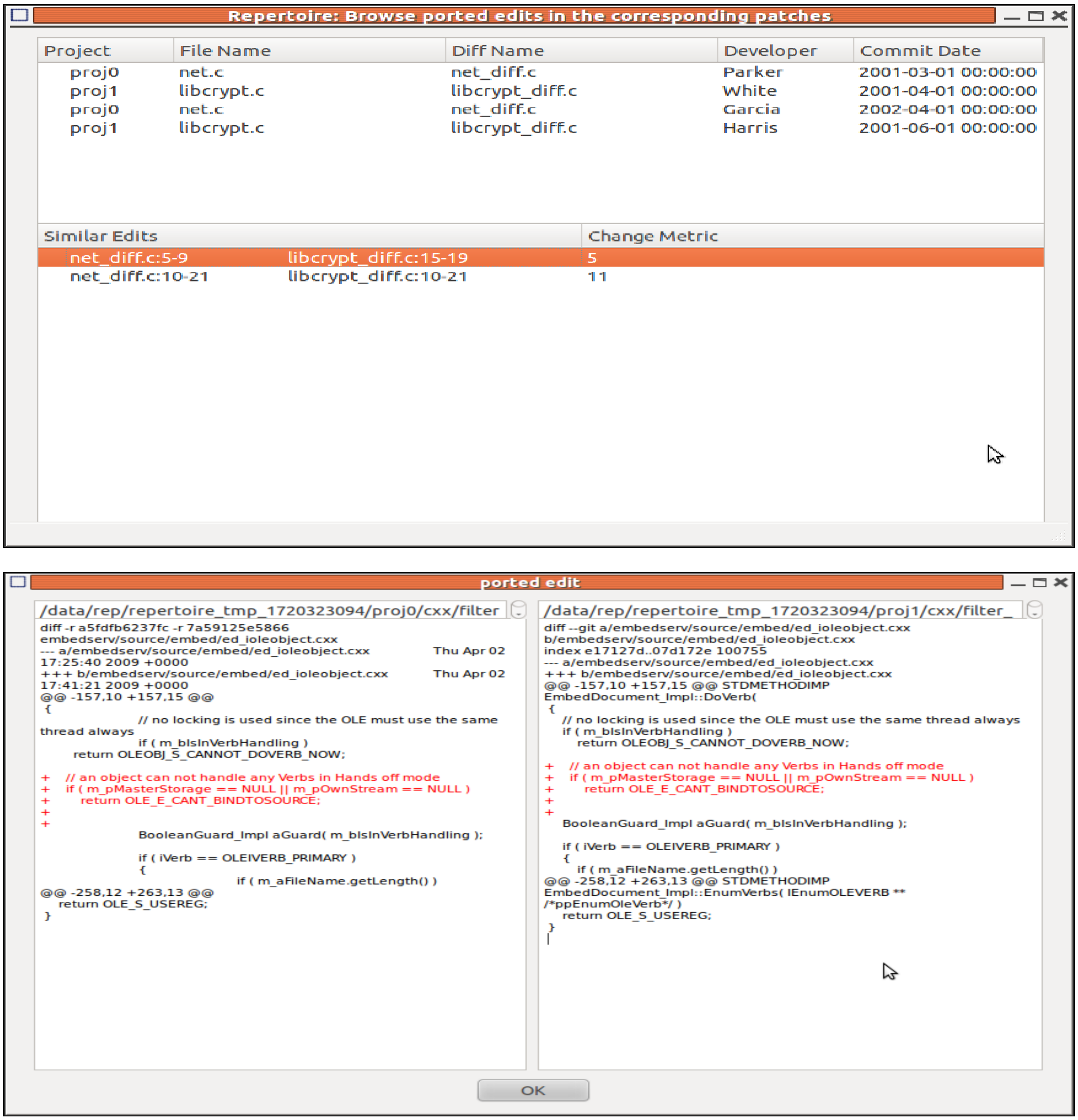


Figure 4: Porting Frequency View. The X-axis is a time line and the Y-axis is the percentage of edited lines in patches ported from other projects.



selected files: proj0:bt.c , proj1:libarch.c

Figure 5: File Distribution View. A point is plotted for each pair of files with ported edits between them.


Repertoire: Browse ported edits in the corresponding patches

Project	File Name	Diff Name	Developer	Commit Date
proj0	net.c	net_diff.c	Parker	2001-03-01 00:00:00
proj1	libcrypt.c	libcrypt_diff.c	White	2001-04-01 00:00:00
proj0	net.c	net_diff.c	Garcia	2002-04-01 00:00:00
proj1	libcrypt.c	libcrypt_diff.c	Harris	2001-06-01 00:00:00

Similar Edits		Change Metric
net_diff.c:5-9	libcrypt_diff.c:15-19	5
net_diff.c:10-21	libcrypt_diff.c:10-21	11

/data/rep/repertoire_tmp_1720323094/proj0/cxx/filter

```
diff -r a5fdfb6237fc -r 7a59125e5866
embedserv/source/embed/ed_ioleobject.cxx
--- a/embedserv/source/embed/ed_ioleobject.cxx          Thu Apr 02
17:25:40 2009 +0000
+++ b/embedserv/source/embed/ed_ioleobject.cxx          Thu Apr 02
17:41:21 2009 +0000
@@ -157,10 +157,15 @@
 {
                // no locking is used since the OLE must use the same
thread always
                if ( m_bIsInVerbHandling )
    return OLEOBJ_S_CANNOT_DOVERB_NOW;

+   // an object can not handle any Verbs in Hands off mode
+   if ( m_pMasterStorage == NULL || m_pOwnStream == NULL )
+       return OLE_E_CANT_BINDTOSOURCE;
+
+
                BooleanGuard_Impl aGuard( m_bIsInVerbHandling );

                if ( iVerb == OLEIVERB_PRIMARY )
                {
                                if ( m_aFileName.getLength() )
@@ -258,12 +263,13 @@
   return OLE_S_USEREG;
 }
```

/data/rep/repertoire_tmp_1720323094/proj1/cxx/filter_

```
diff --git a/embedserv/source/embed/ed_ioleobject.cxx
b/embedserv/source/embed/ed_ioleobject.cxx
index e17127d..07d172e 100755
--- a/embedserv/source/embed/ed_ioleobject.cxx
+++ b/embedserv/source/embed/ed_ioleobject.cxx
@@ -157,10 +157,15 @@ STDMETHODIMP
EmbedDocument_Impl::DoVerb(
 {
    // no locking is used since the OLE must use the same thread always
    if ( m_bIsInVerbHandling )
       return OLEOBJ_S_CANNOT_DOVERB_NOW;

+   // an object can not handle any Verbs in Hands off mode
+   if ( m_pMasterStorage == NULL || m_pOwnStream == NULL )
+       return OLE_E_CANT_BINDTOSOURCE;
+
+
    BooleanGuard_Impl aGuard( m_bIsInVerbHandling );

    if ( iVerb == OLEIVERB_PRIMARY )
    {
       if ( m_aFileName.getLength() )
@@ -258,12 +263,13 @@ STDMETHODIMP
EmbedDocument_Impl::EnumVerbs( IEnumOLEVERB **
/*ppEnumOleVerb*/ )
    return OLE_S_USEREG;
 }
```

Figure 6: The number of ported lines, the corresponding developer, and the dates of the original patch and the ported patch are shown.

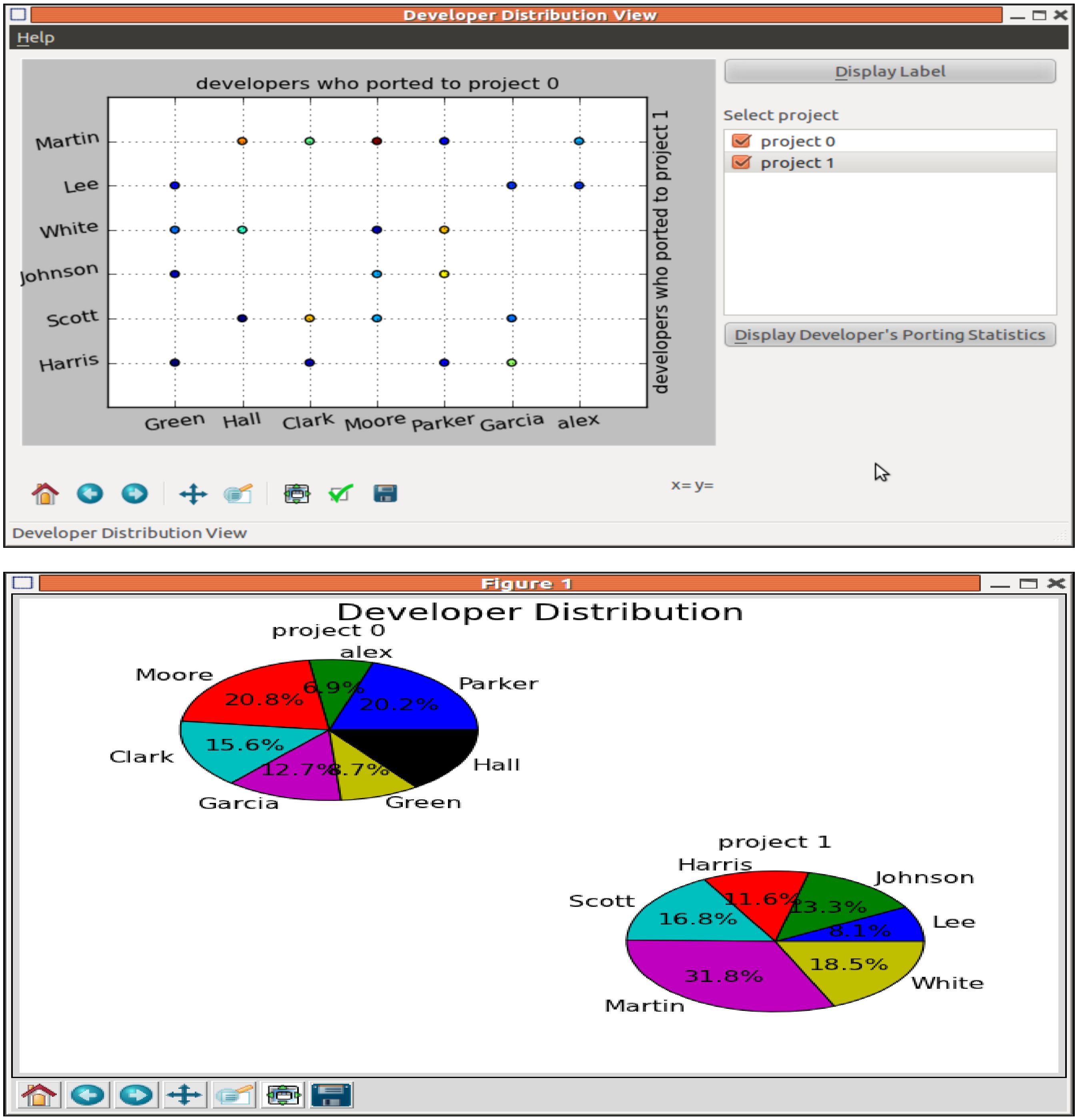


Figure 7: Developer View showing which developers are responsible for what fraction of ported edits.

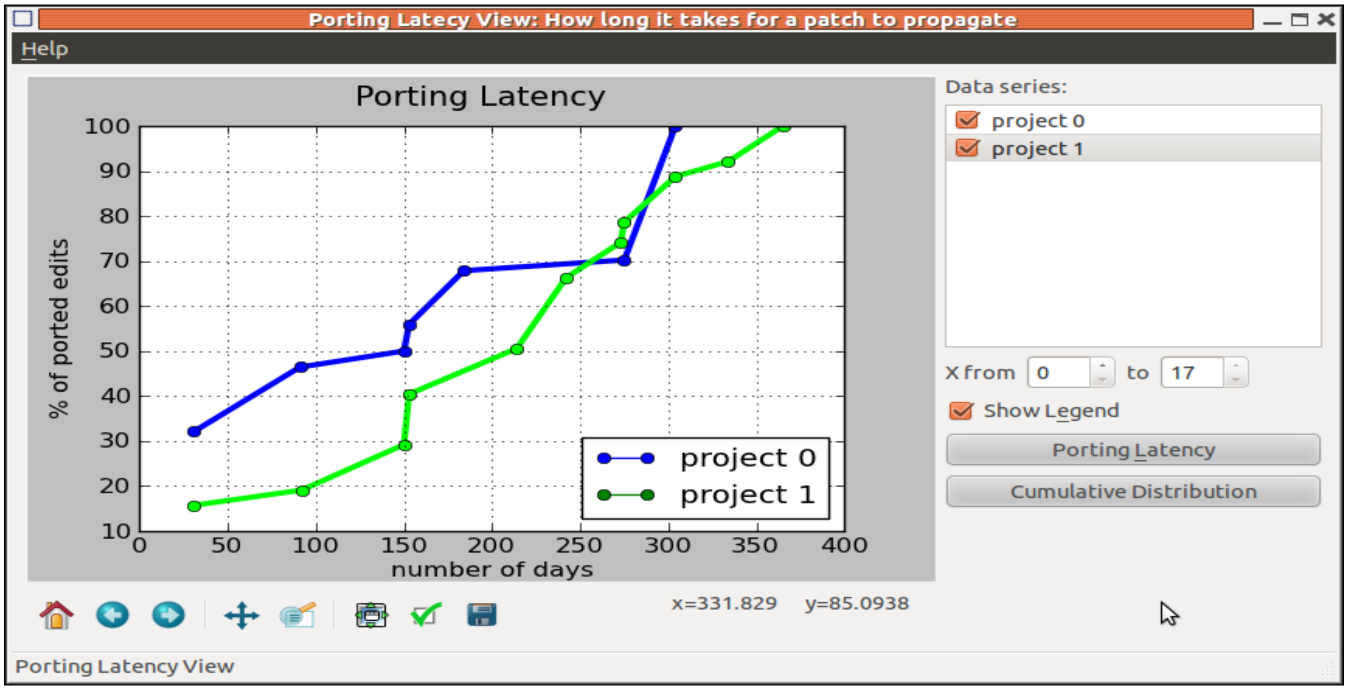


Figure 8: Porting Latency View showing the cumulative distribution of the time taken to port a patch. A porting latency is the time between the commit of an original patch and the commit of a ported patch.