Using Provenance for Repeatability

Quan Pham[1], Tanu Malik[2], Ian Foster[1,2]
Department of Computer Science[1,§] and Computation Institute[2,¶]
University of Chicago[§,¶] and Argonne National Laboratory[¶]
Chicago, IL 60637

Abstract

We present Provenance-To-Use (PTU), a tool that minimizes computation time during repeatability testing. Authors can use PTU to build a package that includes their software program and a provenance trace of an initial reference execution. Testers can select a subset of the package's processes for a partial deterministic replay—based, for example, on their compute, memory and I/O utilization as measured during the reference execution. Using the provenance trace, PTU guarantees that events are processed in the same order using the same data from one execution to the next. We show the efficiency of PTU for conducting repeatability testing of workflow-based scientific programs.

1. Motivation

Scientific progress relies on novel claims and verifiable results. However, testing claims and results described in research papers can be challenging. Increasingly conference committee and journal editors are encouraging authors to submit their code, data, and software for repeatability testing. Repeatability testing improves peer review by allowing reviewers to: (1) not only read the ideas in the paper, but validate them by running the accompanying software; (2) run the software for different data and parameters to check robustness; and (3) determine the limitations and assumptions in the ideas by testing with more general inputs, or under different conditions and environments.

However, as documented by recent experiments, repeatability testing can be arduous and time consuming for both authors and testers [1, 2]. Authors have to prepare code, document it, and make explicit rendering of all dependencies on compilers, operating systems and hardware. For testers, assessing code and data for repeatability can be challenging since documentation is rarely complete and perfect. But more so, as experiments become data and computation intensive, the testing time can be significant [3].

Recently tools have emerged that aid authors and testers in making their software and thus experiments repeatable. CDE helps authors package the code, data, and environment for Linux programs so that they can be run and deployed on other machines without installation or configuration [4]. VisTrails provides provenance support for exploratory computational tasks, maintaining detailed history information about the steps followed in during exploration [5]. RunMyCode.org provides repeatability and workability of computer codes [6] associated with a publication through a companion website. Finally, SOLE allows an author to associate their code and data directly with text phrases in the publication, obviating the need for detailed documentation, yet improving readability and understandability for testers [7].

While these tools are a step towards simplifying repeatability testing for authors, they do not reduce computation and data processing time for repeatability tests. Testers may want to repeat only selected portions of an experiment without having to go through the (often time-consuming) process of repeating the experiment in its entirety. For instance, they may want to avoid running a compute-intensive process that decodes and splits an MPEG-video, or avoid performing a data-intensive text scan, or avoid network communication or transfers. Similarly, they may want to reuse cached results. However, such selective execution of program components is difficult or impossible for the tester, unless appropriate history and provenance information has been captured in a previous reference run.

In this paper, we introduce Provenance-To-Use (PTU), a tool that allows authors to assemble code, data, environment, and provenance of an execution into a single package that can be distributed easily. The resulting package allows testers to view the provenance graph and specify, at a process and file level, nodes of the graph that they want to re-execute, thus saving time and effort for repeatability testing (Section 2). As a demonstration, we show how PTU can be used for repeatability testing of workflow-based programs (Section 3). We conclude in Section 4.

2. PTU: Provenance-to-use

Provenance-To-Use (PTU) [8] is a tool for accelerating and simplifying repeatability testing. Authors can use PTU to accomplish two tasks: (1) build a package of their source code, data, and environment variables, and (2) record process- and file-level details about a reference execution in a database that is included in the package. The resulting package is easily distributed to

testers, who are shielded from software deployment issues. Recording a reference execution path and run-time details within the package eases distribution of this vital information that can guide testers during the testing phase. In particular, testers can explore the provenance graph and accompanying run-time details to specify the part of the provenance graph that they want to re-execute or replay: see Figures 1, 2, and 3.

PTU uses CDE [4] to create and run a package. CDE uses Unix *ptrace* system call interposition to collect code, data files, and environment variables. The *ptrace* mechanism also allows for auditing file and process information, which can be transformed for storing a provenance graph, independent of the application *ptrace*. In PTU, we enhance CDE's use of *ptrace* to store a provenance graph representing a reference run-time execution at the process and file level. We describe how authors and testers can use PTU.

Authors use PTU to create a self-contained package with a reference execution by prepending the application command with the ptu-audit tool as in the following example, which involves the Java application TextAnalyzer applied to a file news.txt:

% **ptu-audit** java *TextAnalyzer news.txt*

The ptu-audit tool uses *ptrace* to monitor ~50 system calls including process system calls, such as *execve*() and *sys_fork()*, for collecting process provenance; file system calls, such as *read*(), *write*(), and *sys_io*(), for collecting file provenance; and network calls, such as *bind*(), *connect*(), *socket*(), and *send*() for auditing network activity. Whenever system calls occur, PTU notes the identifier of the process that made the system call so that it can extract more information about the process from the Linux */proc* file system. In particular, we obtain process name, owner, group, parent, host, creation time, command line, and environment variables; and file name, path, host, size, and modification time. A separate thread is used to obtain memory footprint, CPU consumption, and I/O activity data for each process from */proc/$pid/stat* every three seconds.

In the case of distributed applications involving multiple compute nodes, network activity is audited independently at each node using *ptrace*, i.e., without coordination with other nodes. CDE makes all network system blocking during auditing. Thus, the process record is always present and the information extracted will be current and accurate. PTU stores provenance information about processes and files as a graph in an SQLite database. This information can be displayed in a browser in the form of an Open Provenance Model (OPM) [9] compliant provenance graph.

Currently, PTU's provenance collector makes two assumptions. First, only network connection information is audited and no network dumps are made. Thus, while a tester can replay a computation between exactly the same set of hosts they cannot do so without conducting network communication. Second, PTU does not audit non-deterministic functions such as *ctime()* and *random()*. Relaxing these assumptions to build a general, distributed audit system is part of our ongoing work.

When a system call (file or network) returns, and if a new file or network does not exist, PTU emulates CDE functionality. In the case of a file, it copies the accessed file into a package directory that consists of all subdirectories and symbolic links to the original file's location. In the case of network communication, it saves a log of the network connection information, including IP and port information, ordered by time. The package directory also contains the SQLite database that stores the provenance information of the test run.

When the entire reference run finishes, PTU builds a reference execution file consisting of the topological sort of the provenance graph. The nodes of the graph enumerate run-time details, such as process memory consumption, and file sizes. The tester, as described next, can utilize the database and the graph for efficient re-execution.

Testers can examine the provenance graph contained in a package to study the accompanying reference execution. This graph can be viewed at the granularity of processes and files, and can aid the tester in visually determining parts of the program that they wish to re-execute. For processes, an accompanying bar graph shows CPU and memory consumption. A tester can then request a re-execution, either by specifying nodes in the provenance graph or by modifying a run configuration file that is included in the package. The configuration file initially specifies the provenance graph, corresponding to the reference execution, ordered topologically. A tester can turn flags on or off for each process and file in the provenance graph, to specify if the process needs to be run or if the file needs to be re-generated, respectively.

To re-run the package, testers prepend the program command with a ptu-exec tool as follows:

% **ptu-exec** java *TextAnalyzer news.txt*

The ptu-exec tool uses the provenance graph/run configuration file to determine if any additional process(es) must be run or file(s) re-generated. A re-run of a process or a regeneration of a file is mandatory if:

(1) A process/file is in the descendent sub-graph of another process that is marked for re-running.
(2) A process/file is in the descendent sub-graph of another file that is marked for regeneration.

Re-running these additional processes and regenerating the files is necessary to maintain consistency of the provenance that will be obtained from the test run. To re-execute, ptu-exec obtains run configuration and environment variables for each process from the SQLite database. To re-execute a process, ptu-exec again monitors it via *ptrace* and re-executes CDE functionality of replacing path argument(s) to refer to the corresponding path within the package *cde-package/cde-root/*. By doing so, CDE creates a chroot-like sandbox that tricks the target program into 'believing' that it is executing on the original machine [4]. The following are two of the several ptu-exec command-line options that allow testers to control testing.

- **-time -t1** <t_1> **-t2** <t_2> implies the tester can re-execute everything between t_1 and t_2. If t_1 is null or t_2 is null, execution is performed from beginning to t_2 or t_1 till end, respectively. This option avoids the need to modify the run configuration file.
- **-input** <p> <f_1> <f_2> allows the user to specify input f_1 instead of f_2 for process with id <p>. This option helps user test correctness of the process.

PTU provides fast audit and re-execution independent of the application. The profiling of processes enables testers to choose the processes to run.

3. Use Case and Experiments

To test PTU we took two papers [10, 11] in which authors shared data, tools and software. We constructed meaningful testing scenarios and determined if PTU provides performance improvements. In the first paper, Best et. al. [10] describes $PEEL_0$, a three step workflow process implemented as an R-program. The second step, which executes a classification model, is memory-intensive, with a typical run taking a gigabyte of real memory. To reduce memory consumption, testers may want to input a reduced dataset size and/or test a simpler classification model. (The user can specify the classification model used via a command line option.)

Our second program, TextAnalyzer, is based on the Java-based Unstructured Information Management Architecture (UIMA) and runs a named-entity recognition analysis program using several data dictionaries [11]. It splits the input file into multiple input files on which it runs a parallel analysis. The tester may want to rerun the program on one input split, but with higher convergence criteria. CDE allows testing without installing R and UIMA, as would normally be required,.

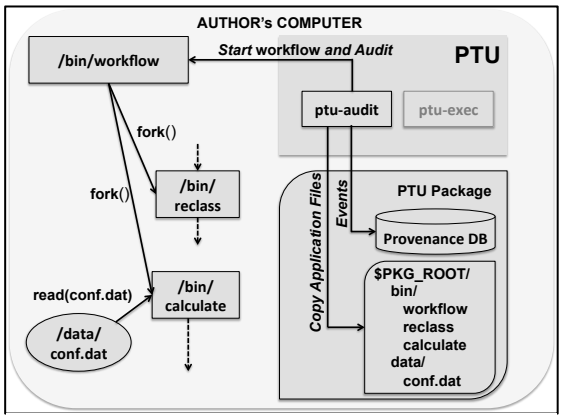

Figure 1. ptu-audit builds a package of provenance events and copies application files

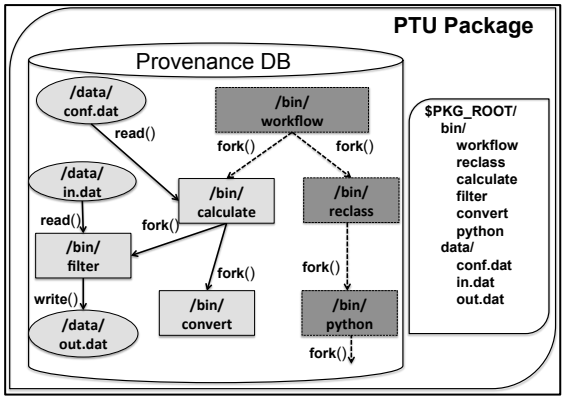

Figure 2. The PTU package. The tester chooses to run the sub-graph rooted at */bin/calculate*

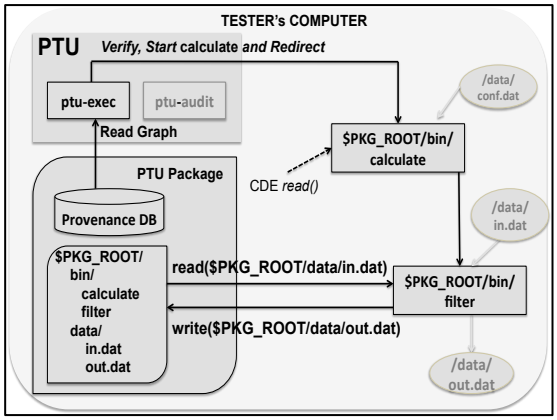

Figure 3. ptu-exec re-runs part of the application from */bin/calculate.* It uses CDE to re-route file dependencies.

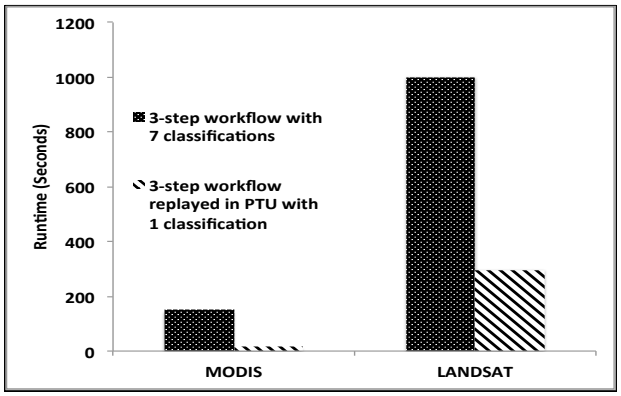

Figure 4. Time reduction in testing PEEL$_0$ using PTU

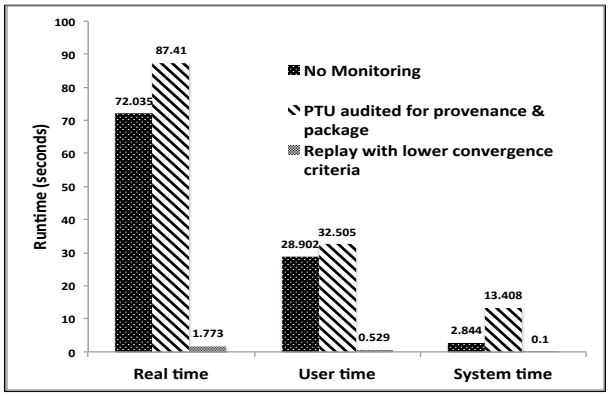

Figure 5. Time reduction in testing TextAnalyzer using PTU

The graphs generated at the process and file level in both programs are large. PEEL$_0$ generates a provenance graph with five process nodes, 10000 exclusive file reads based on the number of files in the dataset, and 422 exclusive file writes for the aggregated dataset. In TextAnalyzer, a single run generates a provenance graph of eight process nodes that in aggregate conduct 616 exclusive file reads, 124 exclusive file writes, and 50 file nodes that are read and written again.

In our experiment, we measure execution times by modifying the PTU configuration file to specify that different inputs should be used for selected processes. For PEEL$_0$, the change involves a different set of input files, and a simplified classification scheme; in the case of TextAnalyzer it is the set of process identifiers to re-execute with a different parameter value. The provenance graphs and configuration files for the programs are available online [8].

Figures 4 and 5 show the performance improvements observed when we thus use PTU to run PEEL$_0$ and TextAnalyzer, respectively, with different parameters. Note first the slow down incurred during the reference executions: ~35% for PEEL$_0$ and ~15% for TextAnalyzer. This slowdown is due to additional system call tracing, with in particular many file system calls traced for PEEL$_0$. These modest increases in execution times are easily offset during the re-execution phase. TextAnalyzer has a particularly large improvement (>98%) since the entire process is run on a much smaller file.

4. Conclusion

PTU is a step toward testing software programs that are submitted to conference proceedings and journals to conduct repeatability tests. Peer reviewers often must review these programs in a short period of time. By providing one utility for packaging the software programs and its reference execution without modifying the application, we have made it easy and attractive for authors to use it and a fine control, efficient way for testers to use PTU.

5. Acknowledgements

We thank Neil Best and Jonathan Ozik for sharing their paper and code. This work was supported by the Center for Robust Decision making on Climate and Energy Policy under NSF grant number 0951576. Contractors of the US Government under contract number DE-AC02-06CH11357 performed the work.

6. References

1. Bonnet, P., et al., *Repeatability and workability evaluation of SIGMOD 2011*. SIGMOD Record, 2011. **40**(2): p. 45-48.
2. Freire, J., et, al. *Computational reproducibility: state-of-the-art, challenges, and DB research opportunities*, *SIGMOD*, 2012.
3. Sasha, D., *Repeatability & Workability for the Software Community: Challenges, Experiences, and the Future*, At: http://www.cs.utah.edu/~eeide/archive10/slides/shasha.pd.
4. Guo, P. et. al., *CDE: Using System Call Interposition to Automatically Create Portable Software Packages*, *USENIX*, 2011.
5. Bavoil, L., et al. *VisTrails: enabling interactive multiple-view visualizations*. In *Visualization*, 2005.
6. Stodden, V., C. Hurlin, and C. Perignon. *RunMyCode.Org*. In *eSoN*, 2012. At: http://www.runmycode.org/CompanionSite/
7. Pham, Q., et al., *SOLE: linking research papers with science objects*, in *IPAW*, 2012.
8. Pham, Q., *PTU: Using Provenance for Repeatability Testing*, 2013. At: http://www.ci.uchicago.edu/SOLE/PTU.html.
9. Moreau, L., et al., *The Open Provenance Model core specification (v1.1)*. FGCS, 2011. **27**(6): p. 743-756.
10. Best, N., et. al., *Synthesis of a Complete Land Use/Land Cover Dataset for the Conterminous United States*. RDCEP Working Paper, 2012. **12**(08).
11. Murphy, J., et. al., *Textual Hydraulics: Mining Online Newspapers to Detect Physical, Social, and Institutional Water Management Infrastructure*, 2013, Technical Report, Argonne National Lab.