A case study in preserving a high energy physics application with Parrot

H Meng[1], M Wolf[2], P Ivie[1], A Woodard[2], M Hildreth[2] and D Thain[1]

[1] Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN 46556, USA

[2] Department of Physics, University of Notre Dame, Notre Dame, IN 46556, USA

E-mail: {`hmeng|mwolf3|pivie|awoodard|mhildret|dthain`}`@nd.edu`

Abstract. *The reproducibility of scientific results increasingly depends upon the preservation of computational artifacts. Although preserving a computation to be used later sounds easy, it is surprisingly difficult due to the complexity of existing software and systems. Implicit dependencies, networked resources, and shifting compatibility all conspire to break applications that appear to work well. To investigate these issues, we present a case study of a complex high energy physics application. We analyze the application and attempt several methods at extracting its dependencies for the purposes of preservation. We propose one fine-grained dependency management toolkit to preserve the application and demonstrate its correctness in three different environments - the original machine, one virtual machine from the Notre Dame Cloud Platform and one virtual machine from the Amazon EC2 Platform. We report on the completeness, performance, and efficiency of each technique, and offer some guidance for future work in application preservation.*

1. Introduction

Reproducibility is a cornerstone of the scientific process [1]. In order to understand, verify, and build upon previous work, one must be able to first recreate previous results by applying the same methods. Historically, reproducibility has been accomplished through painstaking detailed documentation recorded in lab notebooks, which are then summarized in peer-reviewed publications. But as science increasingly depends on computation, reproducibility must also encompass the environment, data, and software involved in each result [2]. It is widely recognized that informal descriptions of software and systems – although common – are insufficient for reproducing a computational result accurately.

In a very abstract sense, reproducing a computation is trivial. Assuming a computation is deterministic, one can simply preserve all the inputs to a computation, then re-run the same code in an equivalent environment, and the same result will be produced. For a small custom application on a modest amount of data, this could be accomplished by capturing the complete environment, data, and software within a single virtual machine image (VMI) [3, 4], and then depositing the image into a curated environment. The publication could then simply refer to the identifier of the image (e.g., DOIs or Amazon Machine Images), which the interested reader can obtain and re-use. A preserved VMI should work on any future platform which supports the hypervisor the VMI depends on. This approach has been used to some success with systems [5].

However, this simple approach is not sufficient for large applications that run in complex environments. There may be *implicit dependencies* on items that are not apparent to the user. For example, the user may understand that his application relies on a particular data analysis package, but would have no reason to know that the package has further dependencies on other libraries. The *granularity* of the dependencies may not be well understood. For example, the user may understand that a computation depends upon a data collection that is 1TB in overall size, but not have detailed knowledge that it only requires three files totalling 300MB out of that whole collection. There may be dependencies upon *networked resources* that are inherently external to the system, such as a database, a code repository [6], or a scalable filesystem [7]. For such resources, it must be decided whether the dependency will simply be noted, or if it must be incorporated whole or in part. Where *common dependencies* are widely used, it may be inefficient or impossible to store one copy of each dependency for each archived object. Some form of sharing or de-duplication is necessary in order to keep the archive to a reasonable size.

We do not claim to have solved all these problems. Rather, our aim in this paper is to highlight the scope of the problems by presenting a case study of one complex application. The application is presented to us first in the form of an email that describes in prose how to install the software and run the analysis. We perform several successive refinements to convert it into an executable and preservable object. We then develop techniques for reducing the size of the dependencies that are necessary for the object to function, and we demonstrate the preserved object functioning correctly in three different physical and cloud environments.

2. Case Study: TauRoast

The application which is the study of this paper is called *TauRoast*. It searches for cases where the Higgs boson produced in association with top quarks decays to two tau leptons. Since the tau leptons and top quarks are very short-lived, they are not observed directly, but by the particle decay products that they generate. So, the analysis must search for detector events that show a signature of decay products compatible with both hadronic tau and top decays. Properties of such events are used to distinguish the events of interest (Higgs decays) from all other events and are also used in further statistical analysis.

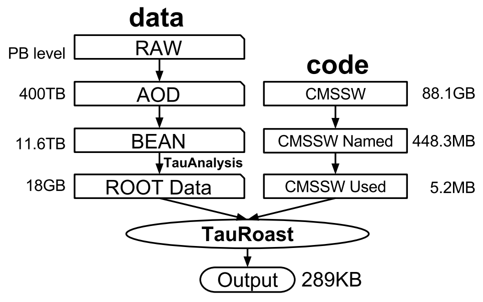

Figure 1. Inputs to TauRoast

Figure 1 shows that both the code and data that form *TauRoast* are drawn from large repositories through multiple steps of reduction. A preservation strategy must weigh whether to store the large repositories completely, the fragments used by an artifact, or something in between.

2.1. Code Sources of TauRoast

Like many scientific codes, the central algorithm of *TauRoast* is expressed in a relatively small amount of custom code developed by the primary author. But, the code cannot run at all without making use of an enormous collection of software dependencies. Some of these dependencies are standard to operating systems worldwide, some are standardized across the entire high energy physics community, some are particular to small collaborative groups, and a few are very specific to a single researcher.

The largest of these repositories is the CMS Software Distribution (CMSSW) [8], a carefully-curated selection of software packages which is distributed in several forms. Historically, components of CMSSW were obtained by checking components of the source out of Concurrent Versions System (CVS), or by installing a complete binary package on a shared filesystem within

Name	*Location*	*Total*	*Named*	*Used*
CMSSW code	CVS	88.1GB	448.3MB	6.3MB
Tau source	Git	73.7MB	73.7MB	6.7MB
PyYAML binaries	HTTP	52MB	52MB	0KB
.h file	HTTP	41KB	41KB	0KB
ROOT data	HDFS	11.6TB	N/A	20GB
Configuration	CVMFS	7.4GB	N/A	103MB
Linux commands	localFS	110GB	N/A	68.4MB
HOME dir	AFS	12GB	N/A	32MB
Misc data	PanFS	155TB	N/A	1.6MB
Total		166.8TB	N/A	21GB

Figure 2. Data and Code Used by TauRoast

an High Performance Computing (HPC) center. In recent years, distribution has moved to an on-demand delivery system known as CVMFS [7], which provides a filesystem interface that transparently accesses a remote repository. The content of CMSSW is managed by a centralized team whose main goal is to ensure that the current version of the software operates correctly on the operating systems and architectures currently in use. However, preservation is not a specific objective of the system, and so there is no particular guarantee that old versions of CMSSW will continue to operate indefinitely.

2.2. Data Sources of TauRoast

The CMS collaboration provides end users with a pre-processed and reduced data format, AOD [9], containing information for events, i.e., proton-proton collisions with a signature of interest, in the form of reconstructed particles. This format is based on the RAW output of the CMS detector readout electronics and reconstructed world-wide, which is then processed through various algorithms which derive signatures of individual particles. Both real and simulated data are available for examination.

As AOD data are too large to be iteratively processed repetitively in an analysis workflow, they are normally reduced further to formats particular to the investigator. In this case study, the AOD data are reduced to BEAN (Boson Exploration Analysis Ntuple) format events, which contain only trivial data containers packed in vectors. This step is performed at Notre Dame by the NDCMS group and is quite CPU intensive, resulting in 11.6 TB to be analyzed by *TauAnalysis*, a small custom code built on top of CMSSW. The BEAN format, production code, and data are shared within the analysis group looking at Higgs production in association with top quarks, which is formed by groups from a few American and European universities, consisting of up to a few dozen contributors.

In the second step, the data are reduced to the ROOT files [10], which contains only events matching basic quality criteria and fields relevant to *TauRoast*. Again, the NDCMS group resources are used to perform this reduction and selection, running highly customized software, built on CMSSW and the BEAN framework, with code written and maintained by a small group.

Once the data has been reduced to ROOT files, *TauRoast* can be run as a single process, and contains a stringent event selection to look only at high quality candidate events for the underlying physical process. Quantities from the relevant events can be both plotted and used in multivariate analysis to determine the level of expected signal in real data. This package is written using the CMSSW build framework, but only utilizes ROOT and a few external Python libraries not found in CMSSW.

3. Observations

The original author of *TauRoast* shared his work through an email which described, in prose, how to obtain the ROOT data through *TauAnalysis*, how to obtain the source code of *TauRoast*, how to build the program, and run it correctly on one specific machine at Notre Dame, with no guarantee that it would work successfully on a new machine. Although this starting point may seem extreme, it is natural for collaborators to share configurations in this form, and to rely on the presence of a working environment already installed.

Starting with this email, the authors of this paper assumed the role of curators, whose job is to prepare the application for permanent archival. First, we elaborated the email instructions into an executable script that declares the necessary environment variables, downloads and checks out the necessary source code, builds it appropriately, calls initialization scripts and then runs the analysis. A few rounds of correction with the original author were necessary to obtain all the dependencies and run the artifact correctly. This process revealed a number of characteristics:

- *Many Explicit External Dependencies. TauRoast* depends on a large number of external dependencies, each with a different access method and data source. While we knew in advance that it depended upon the large CMSSW distribution, it was not apparent until elaborating the script that it depended upon two different Github repositories [11, 12] for the Tau source, a CVS server at CERN for some configuration information, a public website for the PyYAML library [13], and the public home page of a Notre Dame student for one missing header file (the latter is particularly troubling!). While, at some level, the authors and users of these software know of these dependencies, they are often missing in informal communications or forgotten once they are installed.
- *Many Implicit Local Dependencies.* A much harder problem is that the application assumed the presence of many different components in the local filesystem hierarchy. It would be tempting to capture all of these by simply storing a virtual machine image containing the local filesystem. However, the application depends on no less than *five* networked filesystems available on the machine the original author works on: the data to be analyzed was stored on an HDFS [14] cluster, some configuration data was stored on a CVMFS [7] filesystem, and a variety of software tools were on NFS [15], PanFS [16], and AFS [17] systems. The original author was not aware of many of these dependencies, because he relied on local system administrators to provision the machine.
- *Configuration Complexity.* As a means of controlling the complexity of dependent software packages, the HEP community has developed a number of tools that perform run-time configuration and consistency checks of the available software, such as `scram` which is used by CMS. Before running any code, `scram` is used to setup the runtime execution environment, check the availability of every shared library dependency, and build the code. If the correct versions are not available, `scram` halts and emits an error. While this procedure has great value for consistency, it also introduces a significant cost because it involves a large number of nested scripts traversing a filesystem, repeatedly looking up metadata. In our example, the time to perform this configuration check with a cold cache is about 14 minutes, which is almost as long as the actual analysis run of 20 minutes.
- *High Selectivity.* The user's program may mention lots of unused data and software. Often, the program may name a whole data or software repository, but only a handful of items from the repository are really used. For example, the data is stored on an HDFS filesystem with 11.6TB of data, but only 20GB are actually consumed by the program. The reason for this great reduction is at first each BEAN event contains a large amount of information, and *TauAnalysis* throws away a lot of irrelevant event information, keeping only the relevant bits. The CMSSW repository is 88.1GB in total but only 448.3MB of source is checked out, and the actually used software only measures 6.3MB. In a few cases, a source of software is

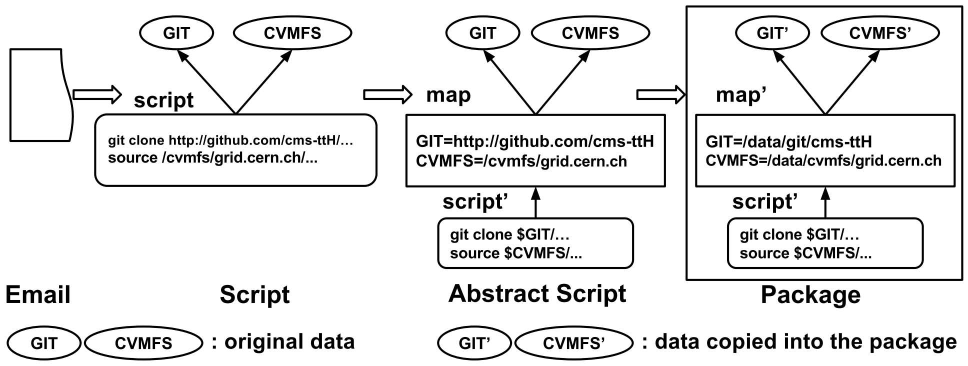
GIT
CVMFS
script
git clone http://github.com/cms-ttH/...
source /cvmfs/grid.cern.ch/...
map
GIT=http://github.com/cms-ttH
CVMFS=/cvmfs/grid.cern.ch
script'
git clone $GIT/...
source $CVMFS/...
GIT'
CVMFS'
map'
GIT=/data/git/cms-ttH
CVMFS=/data/cvmfs/grid.cern.ch
Email
Script
Abstract Script
Package
: original data
: data copied into the package

Figure 3. Version Evolution

named but never actually accessed. We suspect that end users are accustomed to missing dependencies and thus get in the habit of adding commonly used software, whether it is needed or not.

- *Rapid Changes in Dependencies.* Over the process of several months from collecting the initial email into a script until writing this paper, the computing environment continuously changed. The CMSSW software framework released a new version, the target execution node was upgraded to a new operating system, and the CMS community switched from CVS to Git for the management and distribution of the source code. While users seem to be accustomed to these constant changes, any preservation technique must be cautious about relying upon an external service, even one that may appear to be highly stable.

4. Evolving the Artifact

It is clear that the artifact, as provided, is not in a suitable form for preservation. While it might be technically possible to automatically capture the entire machine and all of the connected filesystems into a virtual machine image, it would require 166.8TB of storage (Figure 2), which would be prohibitively expensive for capturing this one application alone. Further, if multiple similar applications are preserved, we would miss the opportunity to identify common dependencies and store them once for multiple artifacts. A more structured approach is needed.

Figure 3 shows how we have evolved this artifact through several stages which make it more suitable for preservation. In each step of evolution, we make the dependencies of the artifact more explicit and available for automated processing. As noted in the previous section, the original author provided us with prose instructions by email which we translated into an executable script. The script has embedded in it a number of external identifiers such as URLs pointing to repositories and paths to networked filesystems. As a general programming practice, embedding such constants in the middle of a program is unwise, and so we extract all of those identifiers and place them outside the script in a *dependency map* or just *map* for short. The dependency map lists all the external dependencies of the application, indicating the type, how they are accessed, and where they are currently located. The resulting *abstract script* then simply refers to abstract file locations such as Git and CVMFS. If properly constructed, the script should not refer to any external resource unless it is indicated in a dependency map.

By extracting the dependencies into a dependency map, we introduce great freedom for the curator to move, transform, and manipulate the dependencies of the artifact without damaging the artifact itself. Given an abstract script and a dependency map, it is straightforward for an automated tool to examine the dependencies in the map, download the missing ones, and then modify the map to point to the local copies of the dependencies. If we group the executable

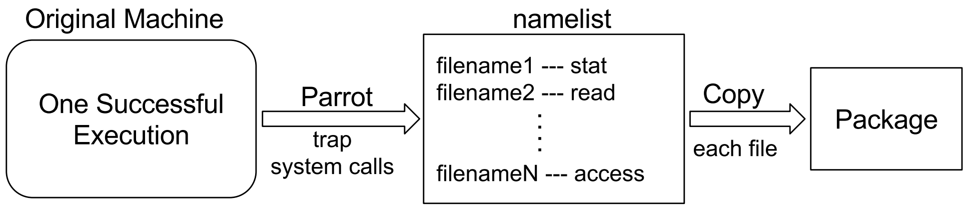

Figure 4. Workflow of The Fine-Grained Dependency Management Toolkit Based on Parrot

script, dependency map, and actual dependencies into a self-contained *package*, we achieve an artifact that can be moved from place to place.

This basic approach to dependency management is a step in the right direction for dependencies that are explicit and external to the user's native execution environment. However, it leaves two other problems unsolved: First, the basic approach requires that someone be *aware* of the dependencies, whether it be the end user, the system administrator, or the archive curator. It seems reasonable to expect the user to be aware of a large dependency mentioned in a top-level script. But, oftentimes the dependency is embedded invisibly deep within the software stack, or is connected to the machine by the system administrator. No single party is likely to have complete information about all of the dependencies. Second, the basic approach assumes that the entire dependency is actually consumed by the artifact. As we have suggested above, this sort of application often only consumes a small fraction of what it does declare as a dependency.

To address both of these problems, users and curators alike need tools that can automatically observe and capture dependencies.

5. Packaging Dependencies with Parrot

We have developed a prototype tool to assist in the measurement and preservation of implicit dependencies for complex applications. We use Parrot [18, 19] to explicitly record all of the files accessed by our example application, allowing us to observe how much of each external dependencies is used, and what local resources are implicitly used. Using this information, we create a *reduced package* which contains only the files actually used by the application.

Parrot is a virtual filesystem access tool which has been used to attach existing programs to a variety of remote I/O systems such as HTTP, FTP, and CVMFS. It works by trapping an application's system calls through the Linux `ptrace` debugging interface, and then replacing them with the desired I/O operations. Parrot is already used in the HEP community to attach applications to the CVMFS distributed filesystem.

Figure 4 illustrates the measurement process. The starting point of this toolkit is one successful execution of an application on the original machine. First, we execute the application under Parrot to generate a *namelist*. From the namelist, we construct a package containing all the necessary data and software for the application. The package effectively becomes a private root filesystem sufficient to run the entire application. It can be re-executed in a variety of ways: Parrot can be used to virtually mount and run the package, chroot can be used to use the package as a root directory, or the package can be converted into a filesystem image for use with a container or virtual machine management system.

Parrot can also be used to track network dependencies. By observing system calls, it can at a minimum record the IP address and port of each external network connection. In addition, it observes the content flowing through each connection and can further infer the protocol and target object of several commonly used protocols. For example, when an application makes an HTTP request, Parrot can record the full URL of the object requested. (however, if the connection is encrypted, this information is not available.)

	*S*hallow *C*opy	*M*edium *C*opy
Whole Files	1632	15642
Empty Files	14273	263
Directories	1549	1549
Symbolic Links	4614	4614
Total Size	21GB	28GB

Figure 5. Package Size

*T*ask *C*ategory	*O*riginal *S*cript	*R*educed *P*ackage
Obtain Namelist	N/A	28min 28s
Generate Package	N/A	26min 19s
Software Acquisition	8min 11s	N/A
Environment Build	5min 49s	4s
Analysis Code	20min 31s	13min 04s

Figure 6. Execution Time

*M*achine Type	*O*S Version	*C*PU Cores	Mem (*G*B)	*E*xecution Time
Original Machine	RHEL 5.10	64	125	13min 04s
KVM (Notre Dame)	CentOS 5.10	4	2	21min 38s
Xen (EC2)	RHEL 5.9	16	60.5	13min 30s

Figure 7. Performance Across Different Machines

For one execution of *TauRoast*, the generated namelist includes 132,047 accessed filenames, along with the system calls used to access the file, such as `open`, `stat`, `read`, etc. With duplicate filenames removed, the namelist is reduced to 67,168 files. Many of those entries do not exist, because they reflect attempts by the application to search for programs and libraries in multiple places. Only 22,068 entries reflect existing files or directories.

The packaging tool iterates over each item of the namelist, determines the process mode and replication degree according to the file type (common files, directories, symbolic links) and the system call type, generates one package containing the dependencies, and summarizes the contents of the package as shown in Figure 5.

We considered several approaches to constructing the package. In a *shallow copy*, we only copied the individual files in the namelist, creating only parent directories for each. Where a directory was listed, we created the directory and populated it with empty files as placeholders to facilitate a directory listing. In a *medium copy*, we copied the individual files as before. Where a directory was listed, we created the directory and copied the contents of the files in that directory, one level deep. A *deep copy* would duplicate all directories recursively, but this would have resulted in TB-sized packages, so we did not consider it further. Generally speaking, the deeper the copy, the larger the package, but the more likely it can be re-purposed.

We evaluated the cost of generating a package and the correctness of the package by running the package and comparing the result to that of running the original script. The results are shown in Figure 6 and 7.

The original script was executed on a 64-core machine running RHEL 5.10 with 125 GB RAM and all the necessary remote filesystems mounted. We measured the time to acquire the software, build the environment, and run the code. Then, using Parrot, we captured the namelist associated with building the environment and running the analysis code *after* running software acquisition. To verify the correctness and portability, we re-ran the package on the original machine, then on an independent KVM virtual machine facility at Notre Dame, and again on a Xen-based virtual machine using the Amazon EC2 platform. While performance varied, the same output was obtained each time, without reference to anything outside the reduced package.

Figure 6 shows the time overhead of this preservation technique. Obtaining the namelist and generating the package increase the execution time as expected, because of the overheads of observing the system calls and copying the necessary files. But once created, the reduced package is considerably *faster* than the original execution, because all of the necessary components are on a single local disk, rather than scattered across distributed resources.

6. Conclusions and Future Work

In this paper, we explore the challenges involved in preserving complex applications by presenting a case study of one high energy physics application, which has many explicit and implicit dependencies and requires a complex execution environment, and propose a fine-grained dependency management solution which can extract exactly all the dependencies of an application and wrap them into a package to be preserved and shared. The preserved package size in our solution is smaller than the virtual machine solution which wraps the whole execution environment into a virtual machine image. We illustrate the feasibility of our solution by preserving all the dependencies of a CMS application into a package and reproducing the application through the preserved package in three different execution environments.

In the following work, we plan to explore how to preserve the input dataset which are shared by lots of CMS applications separate from the software, avoiding preserving multiple copies of the same data in different preserved artifacts, and how to deliver the dataset during runtime efficiently. We also plan to make the structure of the preserved artifacts more clear to make it easier for the new user to extend the original author's work, like testing new input data and adding more analysis code.

Acknowledgments

This work was supported in part by National Science Foundation grants PHY-1247316 (DASPOS), OCI-1148330 (SI2) and PHY-1312842. The University of Notre Dame Center for Research Computing provided critical technical assistance throughout this research effort.

References

[1] Borgman C L 2012 *Proceedings of the 12th ACM/IEEE-CS joint conference on Digital Libraries* pp 19–22
[2] Zabolitzky J G 2002 *Iterations: An Interdisciplinary Journal of Software History* **1** 1–8
[3] Goldberg R P 1974 *Computer* **7** 34–45
[4] Blomer J, Berzano D, Buncic P, Charalampidis I, Ganis G, Lestaris G, Meusel R and Nicolaou V 2014 *Journal of Physics: Conference Series* vol 513 (IOP Publishing) p 032009
[5] Castagné M 2013 *SLIS Student Research Journal* **2** 5
[6] Collaboration C *et al.* 2006 The CMSSW Application Framework
[7] Blomer J, Buncic P and Fuhrmann T 2011 *Proceedings of the first international workshop on Network-aware data management* (ACM) pp 49–56
[8] collaboration C, Acosta D *et al.* 2006 *CERN/LHCC* **1** 421
[9] Holtman K 2001 CMS data grid system overview and requirements Tech. rep. CERN-CMS-NOTE-2001-037
[10] Antcheva I, Ballintijn M, Bellenot B, Biskup M, Brun R, Buncic N, Canal P, Casadei D, Couet O, Fine V *et al.* 2009 *Computer Physics Communications* **180** 2499–2512
[11] CMS ttH TauAnalysis `https://github.com/cms-ttH/ttH-TauAnalysis`
[12] CMS ttH TauRoast `https://github.com/cms-ttH/ttH-TauRoast`
[13] PyYAML `http://pyyaml.org/download/pyyaml/PyYAML-3.10.tar.gz`
[14] Borthakur D 2008 *HADOOP APACHE PROJECT http://hadoop. apache. org/common/docs/current/hdfs design. pdf*
[15] Howard J H, Kazar M L, Menees S G, Nichols D A, Satyanarayanan M, Sidebotham R N and West M J 1988 *ACM Transactions on Computer Systems (TOCS)* **6** 51–81
[16] Welch B, Unangst M, Abbasi Z, Gibson G A, Mueller B, Small J, Zelenka J and Zhou B 2008 *FAST* vol 8 pp 1–17
[17] Sandberg R, Goldberg D, Kleiman S, Walsh D and Lyon B 1985 *Proceedings of the Summer USENIX conference* pp 119–130
[18] Meng H, Kommineni R, Pham Q, Gardner R, Malik T and Thain D 2015 *Journal of Computational Science*
[19] Thain D and Livny M 2005 *Scalable Computing: Practice and Experience* **6** 9–18