A Case Study in Preserving a High Energy Physics Application

Haiyan Meng[1], Matthias Wolf[2], Anna Woodard[2], Peter Ivie[1], Michael Hildreth[2], and Douglas Thain[1]
[1]Department of Computer Science and Engineering, [2]Department of Physics
{hmeng, mwolf3, awoodard, pivie, mhildret, dthain}@nd.edu

UNIVERSITY OF NOTRE DAME

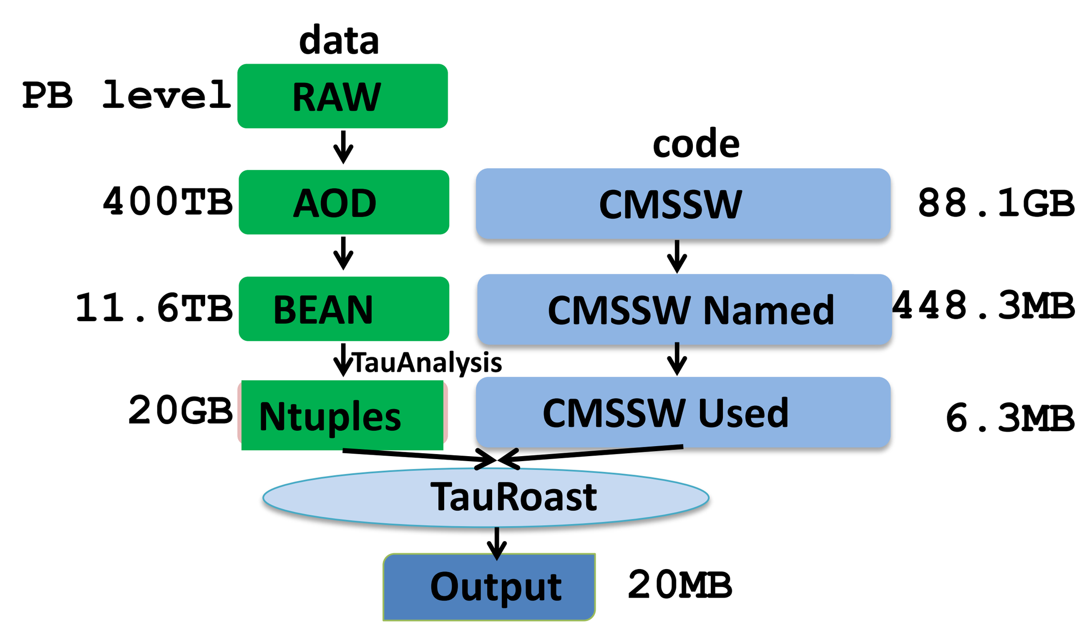
1. ABSTRACT

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. Tools are needed which can automatically identify both local and remote dependencies, so that they can be captured and preserved.

To investigate these issues, we present a case study of preserving a CMS application using Parrot. We analyze the application and attempt several methods at extracting its dependencies for the purposes of preservation. We demonstrate a fine-grained dependency management toolkit which can observe both the local filesystem and remote network dependencies, using the system call tracing capabilities of Parrot. We observe that even a simple TauRoast application depends upon 22,068 files and directories totaling 22 GB of data and software drawn from 8 different sources including CVMFS, HDFS, AFS, Git, HTTP, CVS, PanFS and local root filesystem.

Once the dependencies are observed, a portable execution package can be generated. This package is not tied to any particular technology and can be re-run using Parrot, Docker, a chroot Jail, or as a Virtual Machine Image, depending on the technology available at the execution site. We will report on the performance and completeness of re-execution using both public and private clouds and offer some guidance for future work in application preservation.

2. Input of TauRaust Program



high selectivity of data and code

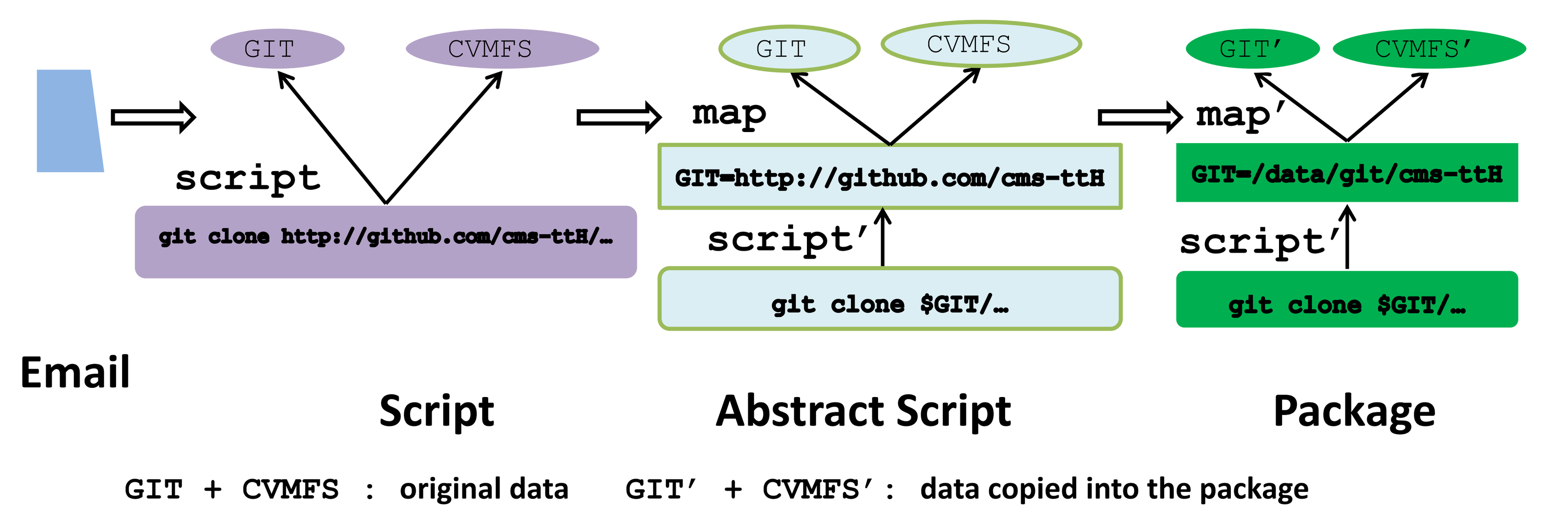
3. Observations

(1) Many Explicit External Dependencies
A) Github repositories for TauRoast source code
B) CVS server for configuration information
C) public web page for the PyYAML library
D) home page of a Notre Dame student for a header file
(2) Many Implicit Local Dependencies
five networked filesystems: HDFS, CVMFS, NFS, PanFS, AFS
(3) Configuration Complexity
hardware, kernel, OS, software , data, and environment variables
(4) Rapid Changes in Dependencies
A) OS upgrades
B) software has newer version
C) CMSSW migrates from CVS to Git
(5) High Selectivity of Data and Software Dependencies

Name	Location	Total Size	Named Size	Used Size
CMSSW code	CVS	88.1 GB	448. 3 MB	6.3 MB
Tau source	Git	73.7 MB	73.7 MB	6.7 MB
PyYAML binaries	HTTP	52 MB	52 MB	0 KB
.h file	HTTP	41 KB	41 KB	0 KB
Ntuples data	HDFS	11.6 TB	N/A	20 GB
Configuration	CVMFS	7.4 GB	N/A	103 MB
Linux commands	localFS	110 GB	N/A	68.4 MB
Home dir	AFS	12 GB	N/A	32 MB
Misc commands	PanFS	155 TB	N/A	1.6 MB
Total		166.8 TB	N/A	21 GB

Table 1: High Selectivity of Data and Software Dependencies

4. Challenge 1: How to redirect data source of each dependency?

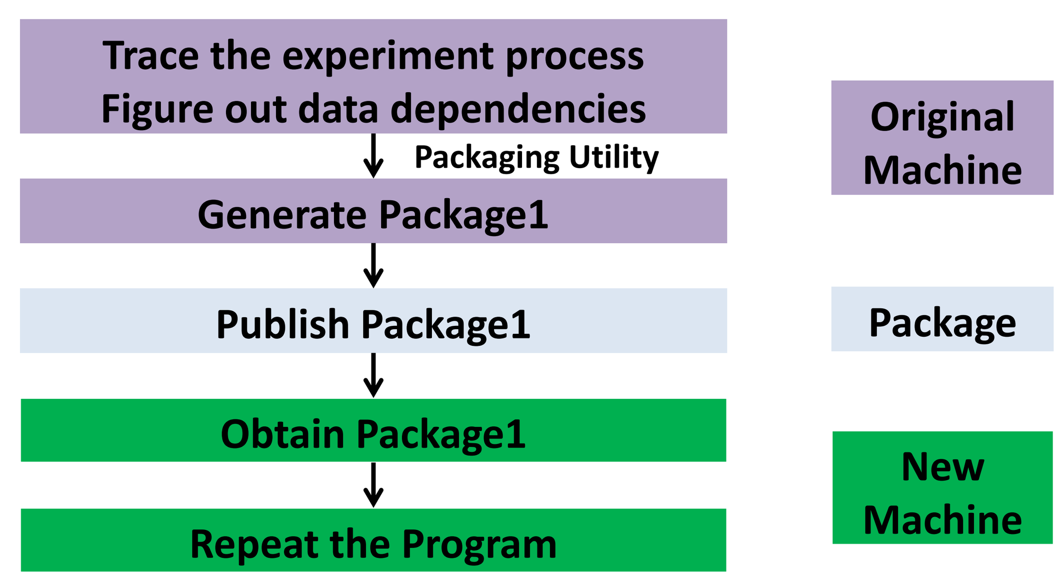


Evolution of Preservation Methods

5. Challenge 2: How to track the used data?

(1) Local dependencies

A) local root filesystem
B) remote filesystems which can be mounted as local directories (CVMFS, HDFS)



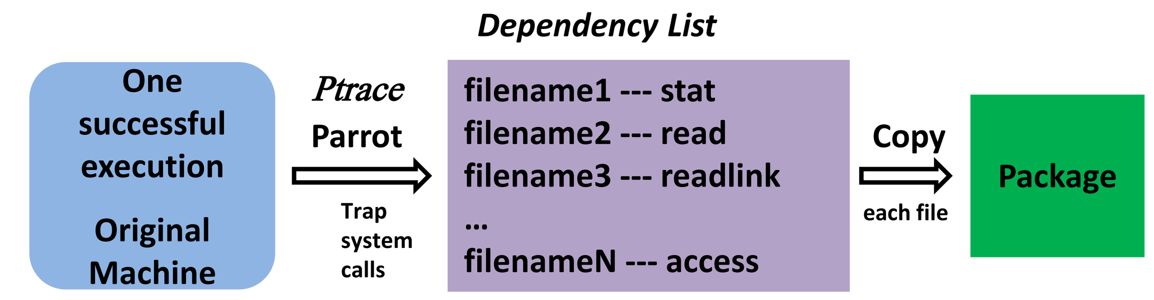
Relationship of Roles

(2) Remote Network dependencies

Aim: evaluate the stability of the network dependencies (Linkrot)
Method: track the network sockets
A) Socket and connect syscalls: the port number, service name (such as, http, https, and ssh), socket type (stream and datagram), and the domain type (inet and inet6);
B) Contents of DNS packets: the hostname and IP address of each remote network dependency;
C) All the http requests and responses.
Problem: as for applications based on https and ssh which encrypt network data using TLS/SSL, tracking network data on the socket level can only see the encrypted data.

6. One Implementation of Package Method

(1) Generate a dependency list from one successful execution
parrot_run --name-list namelist --env-list envlist <user cmd>
(2) Generate a Package containing all the dependencies
parrot_package_create --name-list namelist --env-list envlist --package-path /tmp/package
(3) Repeat one Program within the Package
parrot_package_run --package-path /tmp/package <user cmd>



7. Evaluation

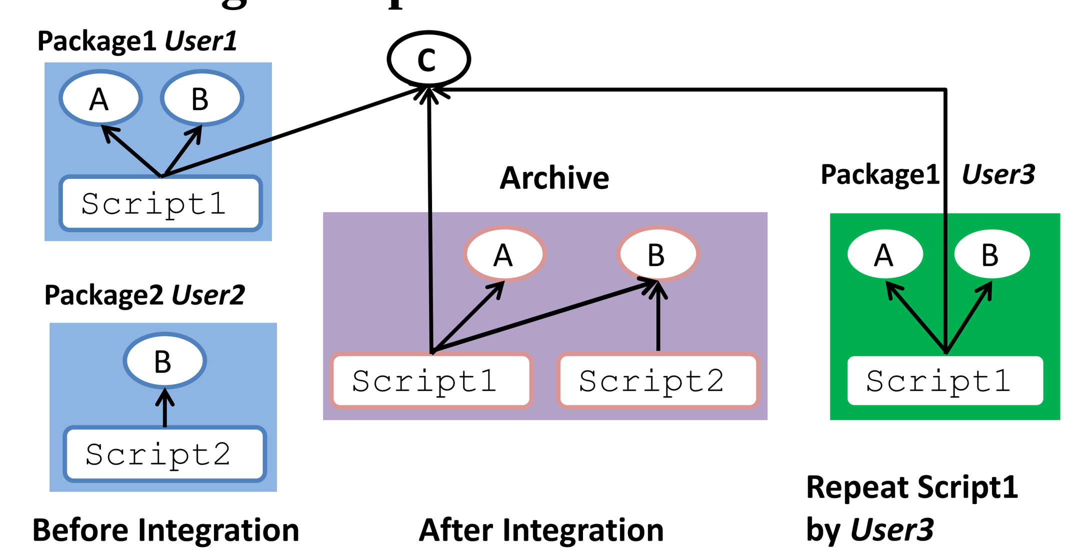
(1) Execution time

Task Category	Original Script	Reduced Package
Obtain Namelist	N/A	28min 28s
Generate Package	N/A	26min 19s
Obtain Software	8min 11s	N/A
Build Environment	5min 49s	4s
Analyze Code	20min 31s	13min 04s

(2) Correctness

Machine Type	Distribution Version	CPU Cores	Memory (GB)	Execution Time
Original Machine	Red Hat 5.10	64	125	13min 04s
KVM (Notre Dame)	CentOS 5.10	4	2	21min 38s
Xen (EC2)	Red Hat 5.9	16	60.5	13min 30s

8. Preserving Multiple Artifacts



9. Open Problems

(1) Measure the Mess or Force Cleanliness?
A) Preserve the whole execution environment into a VMI or package?
B) Specify the execution environment clearly from hardware, kernel, OS, software, data, and environment variables?
(2) Granularity of Dependencies
File? Package? Repository?
(3) Scope of Reuse
A) Exactly repeat what the original other does?
B) Tune the configuration of the original experiment?
C) Change the input data of the original experiment?
(4) Dependency Detection
A) Expert?
B) Tools that can trace the accessed files (e.g., Parrot, CDE)?
C) Package management Tools (e.g., RPM)?
(5) Software Preservation Format
Source code? Binary code? Both?

DASPOS Data and Software Preservation for Open Science

DASPOS: www.daspos.org
Cooperative Computing Lab: ccl.cse.nd.edu
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 scientists and engineers provided critical technical assistance throughout this research effort.