Journal of Computational Science 9 (2015) 137–142

Contents lists available at ScienceDirect

Journal of Computational Science

journal homepage: www.elsevier.com/locate/jocs

An invariant framework for conducting reproducible computational science

Haiyan Meng[b], Rupa Kommineni[a], Quan Pham[a], Robert Gardner[a], Tanu Malik[a,*], Douglas Thain[b]

[a] Computation Institute, University of Chicago, Chicago, IL, USA
[b] Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN, USA

ARTICLE INFO

Article history:
Available online 18 April 2015

Keywords:
Preservation framework
Reproducible research
Virtualization
Container

ABSTRACT

Computational reproducibility depends on the ability to not only isolate necessary and sufficient computational artifacts but also to preserve those artifacts for later re-execution. Both isolation and preservation present challenges in large part due to the complexity of existing software and systems as well as the implicit dependencies, resource distribution, and shifting compatibility of systems that result over time—all of which conspire to break the reproducibility of an application. Sandboxing is a technique that has been used extensively in OS environments in order to isolate computational artifacts. Several tools were proposed recently that employ sandboxing as a mechanism to ensure reproducibility. However, none of these tools preserve the sandboxed application for re-distribution to a larger scientific community aspects that are equally crucial for ensuring reproducibility as sandboxing itself. In this paper, we describe a framework of combined sandboxing and preservation, which is not only efficient and invariant, but also practical for large-scale reproducibility. We present case studies of complex high-energy physics applications and show how the framework can be useful for sandboxing, preserving, and distributing applications. We report on the completeness, performance, and efficiency of the framework, and suggest possible standardization approaches.

1. Introduction

Reproducibility is a cornerstone of the scientific method [4]. Its ability to advance science underscores its importance—reproducing by verifying and validating a scientific result leads to improved understanding, thus increasing possibilities of reusing or extending the result. Ensuring the reproducibility of a scientific result, however, often entails detailed documentation and specification of the involved scientific method. Historically, text and proofs in a publication have achieved this end. As computation pervades the sciences and transforms the scientific method, simple text and static images are no longer sufficient. In particular, apart from textual (and numeric) descriptions describing the result, a reproducible result must also include several computational artifacts, such as software, data, environment variables, platform dependencies and the state of computation that are involved in the adopted scientific method [14].

Virtualization has emerged as a promising technology to reproduce computational scientific results. One such approach is to conduct the entire computation relating to a scientific result within a virtual machine image, and then preserve and share the resulting image. This way "VMI"s become an authoritative, encapsulated, and executable record of the computation, especially computations whose results are destined for publication and/or re-use. Virtual machine images, like files, can then be shared [13]. The resulting image, however, may be too large to share or distribute widely. An alternative light-weight form of virtualization is to encapsulate only the application software along with all its necessary dependencies into a self-contained package. The encapsulation is achieved by operating system-level sandboxing techniques that interpose application system calls and copy the necessary dependencies (data, libraries, code, etc.) into a package, making it lighter weight than a VMI [10]. Yet, the package is not longer an executable record of the computation and still requires an accompanying operating system for execution.

* Corresponding author.
E-mail addresses: hmeng@nd.edu (H. Meng), rupa@uchicago.edu (R. Kommineni), quanpt@uchicago.edu (Q. Pham), rwg@uchicago.edu (R. Gardner), tanum@uchicago.edu (T. Malik), dthain@nd.edu (D. Thain).

http://dx.doi.org/10.1016/j.jocs.2015.04.012
1877-7503/

While both approaches provide mechanisms for encapsulating the computations associated with a scientific result, neither form of virtualization provides any guarantee that the included pieces of software will indeed reproduce the associated scientific result. In general, in the absence of reproducible policy guidelines, such guarantees can be difficult to provide. Preserving the encapsulated computations in such a way that they are always reproducible will improve upon the guarantees. A preservation mechanism can increase the ease of image or package installation, alter dependencies implicit to computation as software components evolve or become deprecated, and provide mechanisms for documentation that make computations easy to understand after the fact.

The two approaches that address the preservation challenge are as follows: one, the introduction of tools that help document dependencies and provide software attribution within VMIs or packages; and two, the use of software delivery mechanisms such as centralized package management, Linux containers, and the more recent Docker framework. We examined the first approach previously in [17]. In this paper we examine the second approach. We consider in particular the lightweight virtualization because we believe together with more standardized software delivery mechanisms, the two combined can address the reproducibility challenge for a wide variety of scientific researchers. A package created by those lightweight approaches encapsulates all the necessary dependencies of an application, and can be used to repeat the application through different sandbox mechanisms, including Parrot [22], CDE [10], PTU [16], chroot, and Docker [3].

Of course our solution represents only one way to preserve applications. Broadly, two different approaches to preserve applications have been adopted: force cleanliness or measure the mess. The former forces users to specify the execution environment for an application in a well-organized way. The latter causes end users to construct the environment as desired, and the complexity of the environment is measured in terms of its dependencies. Our objective here is to measure the mess as-is and then preserve it over time.

To conduct a thorough examination, we consider real-world complex high energy physics (HEP) applications, independently developed by two groups, that must be reproduced so that the entire HEP community can benefit from the analysis. We describe challenges faced in reproducing the applications, and we consider the extent to which reproducibility requirements can be satisfied with lightweight virtualization approaches and software delivery mechanisms. We propose an invariant framework for computational reproducibility that combines lightweight virtualization with software delivery mechanisms for efficiently capturing, invariantly preserving, and practically deploying applications. We measure the performance overhead of lightweight virtualization and software delivery approaches, and show how the preserved packages can be distributed to allow reproduction and verification.

2. High energy physics applications

We study applications taken from two experiments of the CERN Large Hadron Collider, namely the ATLAS experiment and the CMS experiment. In LHC, the ATLAS and CMS experiments are distinct, developed independently by two entirely separate physics communities. Consequently, their applications have very different software distribution and data management frameworks, raising the question of whether common reproducibility frameworks and tools work across the two communities. One of the applications of the ATLAS experiment is the *Athena* application, which is a general purpose processing framework including algorithms for event reconstruction and data reduction [6]. The CMS experiment is conducted through an application termed *TauRoast*, which searches for specific cases where the Higgs boson decays to two tau leptons [8].

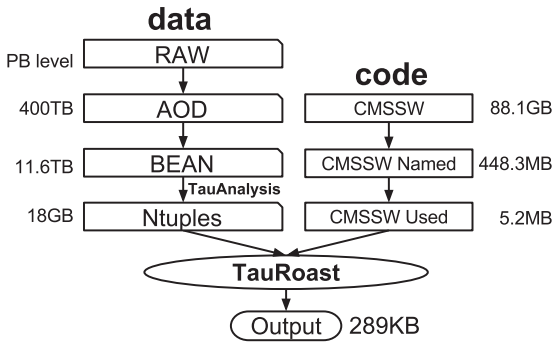

Fig. 1. Inputs to Tau Roast.

Code and data in *TauRoast* are available through five different networked filesystems which are mounted locally, an HDFS cluster for the CMS dataset, some configuration files were stored on CVMFS [2], and a variety of software tools were on an NFS, PanFS and AFS systems. In addition, code may exist in version control systems such as Git, CVS, and CMS Software Distribution (CMSSW).

Data that is input to *TauRoast* is obtained by reducing it through a pipeline, as shown in Fig. 1. Consequently, the real input data may vary depending upon the topic of research. Similarly the software may name many possible components but the used components are smaller than the named ones.

Data in *Athena* is obtained through an external Dropbox-like system called the FaxBox, but does not pass through any reduction steps. Code is obtained through CVMFS, which provides the analysis routines. The invoked configuration will change, however, depending upon the input data code. Thus in Athena the used code and configuration are dynamic depending upon input data, whereas in TauRoast the code and data are static, but the amount of data and code to include changes depending on the science involved.

3. Challenges in reproducing HEP applications

The application specifications of *TauRoast* and *Athena* were provided to us from the CMS and ATLAS Collaborations respectively in the form of email describing in prose how to obtain the source, build the program, and run it correctly on a specific platform type available at our home institutions. There were no explicit guarantees that it would run on alternative platforms. This minimal level of documentation about software is routine in the scientific world. Below we describe the challenges faced when capturing application details in reproducible form and then preserving them for subsequent reuse.

- **Identifying all dependencies.** Due to the distributed, collaborative nature of HEP software development, these applications depend on a large number of external and local software components. External dependencies are often explicitly stated, such as when the application makes connections to Github resources or CVS servers for downloading source files. When the application has initiated execution then implicit network connections may be present that require identification of dependencies on all machines where execution takes place. Implicit local dependencies can arise as a result of mounted filesystems. In *TauRoast*, the application data and code is distributed on five networked filesystems, and in *Athena* on two networked filesystems. Since these filesystems appear local to the application machine, it is important to check and capture mounted filesystems and their respective mount points.