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].

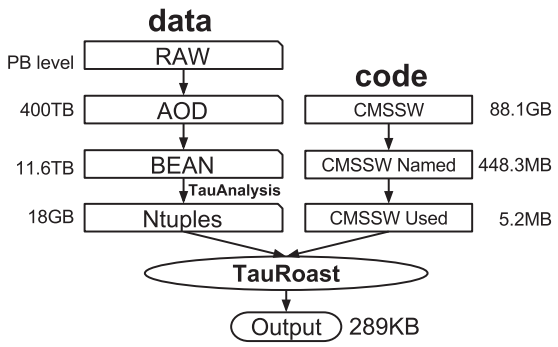

Fig. 1. Inputs to Tau Roast.

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.

- **Configuration complexity.** In order to correctly reproduce an application requires that run-time configurations and consistency checks on the available software are effectively captured and preserved. For CMS, the `scram` software management tool is used to locate the appropriate version of software, set environment variables such as the PATH, run any tool-specific configurations, and do the same for all software on which it depends. A reproducible framework must capture the work of such software management tools so that the framework can conduct similar checks on a new machine.
- **High selectivity.** Although the total size of the resources accessed by HEP programs is very large, the size of the data and software actually used are typically much smaller. The script will often name an entire repository or data source, but the program needs only a handful of items from that source. For example, the data may be stored on an HDFS file system with 11.6TB of data, but the program may consume only 18GB. Thus there is a size tradeoff in terms of capturing dependencies mentioned in the program and dependencies actually used in the program. A reproducible framework must include robust rules about not including superfluous dependencies, but including unused dependencies that may potentially see much use during program execution.
- **Rapid changes in dependencies.** Over the course of three months between collecting the initial email, analyzing the programs, and writing this paper, the computing environment saw continuous change. The CMSSW software distribution released a new version, the target execution environment was upgraded to a new operating system, and the application switched from CVS to Git for obtaining the software. For Athena, the computing environment has the potential for daily change since upgrades to the software framework occur on a nightly basis. While the users of this software seem accustomed to constant change, which is appropriate during algorithm development, any new technique for preservation that may rely on an external service (every one that may appear highly stable) will require caution to choose the actual releases used during the time of publication.
- **Dependencies for reproducible execution.** Capturing the necessary and sufficient dependencies that are part of an application is sufficient for repeatability, but possibly for not reproducibility. Repeatability implies that if a result depends on running program X, we must be able to run exactly X again. In reproducibility the goal is rarely limited to running *precisely* what a predecessor did. Often, the objective is to change a parameter or a data input in order to see how the result is affected. To that end, the preservation system must capture enough of the surrounding material in order to permit modifications to succeed. Further, a better understanding of how end users will consume preserved software will help to shape how software should be preserved.

4. The invariant framework

Given the many challenges of reproducing HEP applications, we now describe an invariant framework. If present within large collaborations, this framework can enable application developers to share their application with other researchers, and for other researchers to reproduce the shared application. To satisfy invariance, the framework must include mechanisms for:

- **Capturing dependencies and configurations:** Capturing tools must record dependencies that are used by the program, including hardware, OS, kernels, static and dynamic dependencies, local and networked dependencies, source codes and data files. Stateful interactions with commercial software, such as proprietary databases and which cannot be captured due to licensing agreements must persist such that replaying later may be accomplished without the presence of the commercial software. In effect, a captured application should behave in exactly the way the application developer intended.
- **Preservation of captured entities:** By preservation we define appropriate mechanisms for (a) documentation of the application development, and (b) automation of any task that becomes necessary to the repetition of the application in exactly the the way the application developer intended.

  Documentation and specification during application development can be onerous. The preservation framework must make programming tools available that focus less on documentation, and more on scripts, integration, and execution of the dependencies such that they are resolved as part of documentation. Automation can extend to various tasks necessary for ensuring repeatability such as building software, provisioning of hardware, validation of software against security fixes, new features, and even monitoring the reproducibility state of a preserved application, i.e., its source code, dependencies, environment, and platform. Automated builds and provisioning and continuous integration service can significantly lower the barriers to running applications in a new environment.

  Despite preservation mechanisms, the application software may not run as intended. For a reproducers understanding, it may also be useful to include a *logical preservation unit* (PLU) that consists of a minimal execution of the software using a small, test data sample and with specified outputs. The provenance of this PLU must be captured so that the reproducer can compare the current run with future reproduction-validation runs.
- **Distribution of preserved packages:** A captured and preserved application must be persistently stored and distributed through a repository. We imagine these repositories to be themselves preserved, and linked with a digital library. Metadata and flexible annotation should be part of this repository for curation over time.

5. Component tools for reproducible research

We have constructed a first approximation of the invariant framework by using and modifying a variety of existing technologies. Of course, a viable long-term strategy for reproducibility must not depend on a single technology. To this end, we have identified multiple technologies that can implement each stage of the framework.

Capturing dependencies. The Unix `ptrace` mechanism allows a tracing process to observe all of the system calls performed by an application, inferring each of the resources upon which an application depends. We have extended two existing tracing tools in order to capture dependency information at the granularity of files and directories. Dependency may refer to source code, if available, or a binary file.

Parrot is a virtual filesystem access tool which is used to attach applications to a variety of remote I/O systems such as HTTP, FTP, and CernVM-FS. It works by trapping system calls through the `ptrace` interface and replacing selected operations with remote accesses. As a side-effect, Parrot is also able to modify the filesystem namespace in arbitrary ways according to user needs. Parrot is particularly used in the high-energy physics community to provide remote access to application software via CernVM-FS. To capture dependencies, we made small modifications to Parrot to record the logical name of every file accessed by an application into an external dependency list. After execution is complete, a second tool is used to copy all of the named dependencies into a package.

The second tool, PTU [18], is designed to create a package of an application by recording all of the binaries, libraries, scripts, data files, and environment variables used by a program. PTU uses the

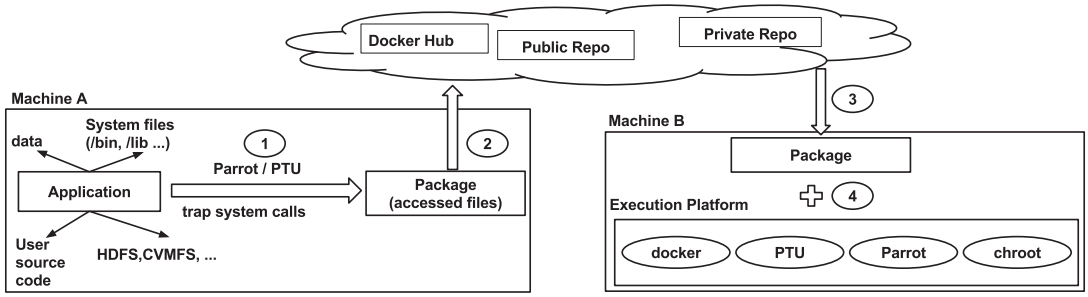

Fig. 2. Preservation framework.

lightweight virtualization software tool Code Data Environment (CDE) to observe system calls, but takes a snapshot of every file at the point of access. In addition to files, PTU records *metadata* about the execution environment, such as Linux kernel version, application version, and dynamic library versions by using standard Unix commands. PTU also records *provenance* in the form of a graph that describes how each file is created or consumed by processes within the application. Because PTU is focused solely on the problem of preservation, it can achieve lower overhead than Parrot when remote data access is not a requirement, as we show below.

Preservation of dependencies. Both Parrot and PTU can observe the precise set of files accessed by an application. If this precise list of files is preserved then it should be possible to execute *exactly* the same application on *exactly* the same inputs a second time. However, if the objective is to *re-purpose* the application by running it in slightly different configurations, then the preserved package may need to be more comprehensive than the strict dependency list.

Many possible heuristics exist for creating the preserved package from the dependency list. We have implemented three: In a **shallow copy**, we copy only the exact dependencies. Where a directory was listed, a directory is created and populated with empty files as placeholders to facilitate a directory listing. In a **medium copy**, every file in every directory mentioned by the application is preserved one level deep. In a **deep copy**, every file in every directory mentioned by the application is copied recursively. Obviously, the more aggressive the preservation, the larger the package, but also the greater the possibility that the package can be adapted to other uses. Other approaches to package generation might including generating the union of multiple dependency sets, or allowing an expert user to manually add and remove dependencies.

Both Parrot and PTU generate packages that consist of plain filesystem trees representing the namespace and data of the preserved application, and can be easily transformed into whatever archive format (e.g. ZIP or TGZ) is most suitable. This is desirable for long-term preservation that may outlive various deployment technologies. However, neither technology yet integrates the programming environment envisioned above for the documentation purpose. Without these techniques, the preserved packages will have diminishing value over time.

Distribution and deployment. Once generated, application packages must be collected, curated, and made available through publicly shared repositories. Currently, a wide variety of services and efforts exist in order to share binary objects in this way, and so we assume such multiple services will be available in the future (Fig. 2).

Of more immediate interest is the ability to deploy a preserved package at a desired execution site. Again, long-term preservation requires artifacts that are independent of any particular technology, so the package must be sufficiently self-describing in order to work with multiple technologies. The packages produced by both Parrot and PTU can today be re-executed through any of the following mechanisms:

(1) Re-running the application through Parrot, which can dynamically construct the desired namespace and limit file accesses to within the preserved package. (2) Generating a virtual machine image from the package, which can be loaded into a local virtual machine monitor, or transferred to a cloud service provider. (3) Converting the package into a Docker image format, enabling it to be deployed within a lightweight Linux container.

Linux containers and Docker images. Linux Containers provide multiple isolated instances of execution on top of the same kernel through OS-level virtualization. Thus they can be used to persist the captured packages into images. Using such containers, Docker now allows for the preservation of the image in a more user-friendly way. Docker also provides portable deployment of containers across platforms, documentation of packages in a scriptable format, and versioning of container images. The image can be preserved along with *dockerfiles*, which are text files containing all the commands to build a Docker image. Similar to shell scripts, the computational section can help other provisioning tools (e.g. Chef, Puppet). The text section, written for human consumption, is more suited for use with a version management system such as subversion or git, which can track any changes made to the dockerfile. Thus dockerfiles can be used to preserve the namelist of Parrot packages and provenance description in PTU. Docker is integrated with a continuous build environment that will not only check and validate the version of the software in present use, but also make use of a more recent version to build application software.

6. Evaluation

We evaluated the correctness and performance of running, packaging, and re-running the Athena and TauRoast applications using the Parrot and PTU tools. To do this, each application was first executed directly and its execution time and output were recorded. Then the application was executed under Parrot and PTU, and a self-contained package was created for each case. Finally, the application was re-executed using the package. The time overhead of each execution and re-execution was collected and compared with the recorded reference.

The TauRoast application checks and evaluates a dataset with the size of 18 GB stored in an HDFS file system, and can be finished in about 20 minutes when running directly on a server with 64 cores and 126 GB memory. The output of the application includes an event analysis log and a statistical information, and its size is approximately 289 KB. The Athena application processes a given input data file to produce four “derived” data files whereby event selection corresponding to interest physics channels are made. It uses the nightly release of the Athena framework and submits an analysis (“event skimming”) job to obtain derived data, a common use case in high energy physics data analysis.

Table 1 compares the time overhead of preserving Athena and TauRoast application using Parrot and PTU. Parrot splits the packaging creation procedure into two sequential steps: first, execute the application within Parrot and generate the accessed file namelist;

Table 1
Time comparison between parrot and PTU.

Application Name	Packaging Tool	Obtain Name list	Create Package	Re-Execution (Tool/Time)
Athena	Parrot	10 min 14 s	00 min 53 s	Parrot / 09 min 14 s
Athena	PTU	–	08 min 48 s	PTU / 07 min 21 s
TauRoast	Parrot	22 min 50 s	04 min 25 s	chroot / 10 min 24 s
TauRoast	PTU	–	23 min 30 s	PTU / 08 min 40 s

second, traverse the namelist and copy all the accessed files into a self-contained package. PTU accomplishes the execution procedure and the package creation procedure concurrently through multi-threading, bringing 17.5% additional time overhead.

The re-execution time is half or less than half of the original execution time in both cases of the *TauRoast* application. During the original execution, the input dataset is either locally available or comes from HDFS, which is accessed through FUSE kernel modules. During the package creation procedure, all the input dataset has been copied from HDFS into the package on the local filesystem. Thus the re-execution procedure can obtain its input from the local filesystem instead of reading the input dataset from HDFS.

Both packages created by Parrot and PTU are a subset of the root filesystem, which only includes all the accessed files. The original relationship, such as symbolic links between files and directories is maintained. The files from pseudo filesystems such as proc and dev, are ignored. The re-execution procedure uses these pseudo filesystems from the host machine through redirection techniques. For the *TauRoast* application, the sizes of the packages created by Parrot and PTU are nearly the same, 18 GB. Except for the accessed files, both Parrot and PTU preserve the execution environment of the application. The PTU package also includes a leveldb-format provenance information of the application with the size of 3 MB.

Table 2 illustrates the total size and actually used size of each source for TauRoast. The total size is too large to be put into a separate image. However, the actually used size is greatly reduced to be 18 GB. Within the package, the input dataset nearly occupies the whole package. All the other libraries and software dependencies only occupies about 200 MB. As the input dataset grows, it can be put outside the package to reduce the shipping time of the package.

For the Athena application example, the input file was 224 MB and output files were 16 MB each. PTU builds a 855 MB package, while Parrot builds an 825 MB package. The overhead is of provenance which is 7 MB and some differences is dependency information.

7. Related work

The capture and preservation environments were treated as one entity in [15,11]. However, frequently changing experiment software makes the maintenance of the captured experimental environment very complex. CernVM [5] treated them as two different categories. The capturing of computing environment is implemented within CernVM, and the preservation of software environment is based on a CernVM filesystem(CVMFS) specifically designed for efficient software distribution. In fact CVMFS [5] published pre-built and configured experiment software releases to avoid the time-consuming software building procedure, i.e., it did not preserve software in source code format as emphasized in [7]. However, as we show a simple VMI of binaries can also be too big in size for distribution, and the preservation itself needs to include a documentation stage and a distribution stage. We have described capture tools that include software code when available to be included in the package.

Attempts from different perspectives to facilitate the reproduction of scientific experiments utilizing a preserved software library have been made. The software distribution mechanism over network was discussed in [9,2]. A distribution hub through the integration of user interface, scientific software libraries, knowledge base into problem-solving environment was described in [19]. The creation and distribution of language-independent software library by addressing language interoperability was proposed in [12]. A scalable, distributed and dynamic workflow system for digitization processes was proposed in [20]. A distributed archival network was designed in [21] to facilitate process-oriented automatic long-term digital preservation. The work in [1] aimed to help non-domain users to utilize the digital archive system.

Table 2
Data and code size used by TauRoast.

Dependency Name	Location	Total Size	Used Size
CMSSW code	CVS	88.1 GB	5.2 MB
Tau source	Git	73.7 MB	212 KB
PyYAML binaries	HTTP	52 MB	0 KB
.h file	HTTP	41 KB	0 KB
Ntuples data	HDFS	11.6 TB	18 GB
Configuration	CVMFS	7.4 GB	105 MB
Linux commands	localFS	110 GB	110 MB
HOME dir	AFS	12 GB	24 MB
Misc commands	PanFS	155 TB	8 KB
Total		166.8 TB	18 GB

8. Conclusions and future work

In this paper, we propose an invariant framework for conducting reproducible computational science - using light-weight virtualization approaches to preserve applications in the format of self-contained packages and using standardized software delivery mechanisms to deliver and distribute preserved packages. We use two complex high energy physics applications to illustrate how the framework can help the original authors preserve and distribute the applications, and others reproduce the applications.

This paper focuses on how to measure the mess and track the used dependencies to preserve an application. In the following work, we plan to explore how to preserve an application in an organized style - specifying the execution environment clearly. How to preserve and improve the availability of remote network resources is another important problem to be explored.

The DOI name for the experiment involved in the paper is doi:10.7274/R0C24TCG, and current information may be found on the web through http://doi.org/10.7274/R0C24TCG

The Athena experiment is further preserved at: https://sites.google.com/site/invariantcompatlas/

Acknowledgments

This work was supported in part by National Science Foundation grants PHY-1247316 (DASPOS), OCI-1148330 (SI2), PHY-1312842, ICER-1440327, SES-0951576 (RDCEP), and ICER-1343816 (UChicago subcontract). The University of Notre Dame Center for Research Computing scientists and engineers provided critical technical assistance throughout this research effort. The Open Science Grid at the University of Chicago provided critical technical assistance throughout this research effort.

References

[1] Maristella Agosti, Nicola Orio, To envisage and design the transition from a digital archive system developed for domain experts to one for non-domain users, in: Proceedings of the 12th ACM/IEEE-CS Joint conference on Digital Libraries, ACM, 2012, pp. 11–14.

[2] Jakob Blomer, Predrag Buncic, Thomas Fuhrmann, CernVM-FS: delivering scientific software to globally distributed computing resources, in: Proceedings of the First International Workshop on Network-Aware Data Management, ACM, 2011, pp. 49–56.

[3] Carl Boettiger, An introduction to Docker for reproducible research, ACM SIGOPS Oper. Syst. Rev. 49 (1) (2015) 71–79.

[4] Christine L. Borgman, Data, data use, and scientific inquiry: Two case studies of data practices, in: Proceedings of the 12th ACM/IEEE-CS Joint Conference on Digital Libraries, 2012, pp. 19–22.

[5] P. Buncic, C. Aguado Sanchez, J. Blomer, L. Franco, A. Harutyunian, P. Mato, Y. Yao, CernVM-a virtual software appliance for LHC applications, in: Journal of Physics: Conference Series, vol. 219, IOP Publishing, 2010, p. 042003.

[6] P. Calafiura, M. Marino, C. Leggett, W. Lavrijsen, D. Quarrie, The athena control framework in production, new developments and lessons learned, 2005.

[7] Michel Castagné, Consider the source: the value of source code to digital preservation strategies, SLIS Stud. Res. J. 2 (2) (2013) 5.

[8] Serguei Chatrchyan, V. Khachatryan, A.M. Sirunyan, A. Tumasyan, W. Adam, T. Bergauer, M. Dragicevic, J. Erö, C. Fabjan, M. Friedl, et al., Search for the standard model higgs boson produced in association with a top-quark pair in pp collisions at the lhc, J. High Energy Phys. 2013 (5) (2013) 1–47.

[9] G. Compostella, S. Pagan Griso, D. Lucchesi, I. Sfiligoi, D. Thain, CDF software distribution on the grid using parrot, in: Journal of Physics: Conference Series, vol. 219, IOP Publishing, 2010, p. 062009.

[10] Philip J. Guo, Dawson R. Engler, CDE: using system call interposition to automatically create portable software packages, in: USENIX Annual Technical Conference, 2011.

[11] N. Chue Hong, S. Crouch, S. Hettrick, T. Parkinson, M. Shreeve, Software Preservation Benefits Framework, Software Sustainability Institute Technical Report, 2010.

[12] Scott R. Kohn, Gary Kumfert, Jeffrey F. Painter, Calvin J. Ribbens, Divorcing language dependencies from a scientific software library, PPSC (2001).

[13] Sotiria Lampoudi, The path to virtual machine images as first class provenance, Age, 2011.

[14] Tanu Malik, Quan Pham, Ian T. Foster, SOLE: towards descriptive and interactive publications, CRC Press, 2014.

[15] Brian Matthews, Arif Shaon, Juan Bicarregui, Catherine Jones, Jim Woodcock, Esther Conway, Towards a methodology for software preservation (2009).

[16] Quan Pham, Tanu Malik, Ian T. Foster, Using provenance for repeatability, in: USENIX NSDI Workshop on Theory and Practice of Provenance (TaPP), 2013.

[17] Quan Pham, Tanu Malik, Ian T. Foster, Auditing and maintaining provenance in software packages, in: International Provenance and Annotation Workshop (IPAW), 2014.

[18] Quan Tran Pham, A Framework for Reproducible Computational Research, PhD thesis, The University Of Chicago, 2014.

[19] John R. Rice, Ronald F. Boisvert, From scientific software libraries to problem-solving environments, IEEE Comput. Sci. Eng. 3 (3) (1996) 44–53.

[20] Hendrik Schöneberg, Hans-Günter Schmidt, Winfried Höhn., A scalable, distributed and dynamic workflow system for digitization processes, in: Proceedings of the 13th ACM/IEEE-CS Joint Conference on Digital Libraries, ACM, 2013, pp. 359–362.

[21] Ivan Subotic, Lukas Rosenthaler, Heiko Schuldt, A distributed archival network for process-oriented autonomic long-term digital preservation, in: Proceedings of the 13th ACM/IEEE-CS Joint Conference on Digital Libraries, ACM, 2013, pp. 29–38.

[22] Douglas Thain, Miron Livny, Parrot: an application environment for data-intensive computing, Scalable Comput. Pract. Exp. 6 (3) (2005) 9–18.