An Invariant Framework for Conducting Reproducible Computational Science

Haiyan Meng[2], Rupa Kommineni[1], Quan Pham[1]
, Robert Gardner[1], Tanu Malik[1], and Douglas Thain[2]

[1] Computation Institute, University of Chicago, Chicago, Illinois, USA
`rupa, quanpt, rwg, tanum@uchicago.edu`
[2] Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, Indiana, USA
`hmeng, dthain@nd.edu`

Abstract
Computational reproducibility depends on being able to isolate necessary and sufficient computational artifacts and preserve them for later re-execution. Both isolation and preservation of artifacts can be challenging due to the complexity of existing software and systems and the resulting implicit dependencies, resource distribution, and shifting compatibility of systems as time progresses—all conspiring to break the reproducibility of an application. Sandboxing is a technique that has been used extensively in OS environments for isolation of 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 combined sandboxing and preservation framework, which is efficient, invariant and 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.

Keywords: Preservation framework, reproducible research, virtualization, container

1 Introduction

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

Virtualization has emerged as a promising approach for reproducing computational scientific results. One approach is to conduct the entire computation relating to a scientific result within a virtual machine image, and then share the resulting image. This way VMIs become an authoritative, encapsulated, and executable records of computations, especially computations whose results are destined for publication and/or re-use, and the VMIs can be shared easily [13]. However, the resulting image may be too big to distribute. An alternative light-weight form of virtualization allows encapsulation of the application software, along with all its necessary dependencies into a self-contained package, but does not make it executable. The encapsulation of the self-contained package is achieved by interposing application system calls, and copying the necessary dependencies (data, libraries, code, etc.) into the package, making it lighter weight than a VMI [10]. While both approaches provide mechanisms for sandboxing 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.

Since reproducibility includes documentation, virtualization approaches in their current form only make it easy to capture the computations. Preserving the computations so that they are easy to understand, install, or alter implicit dependencies that are part of computation is not effectively addressed, especially as dependencies and software components evolve or become deprecated. There are two approaches to address the preservation challenge. By either introducing tools that help document dependencies and provide software attribution within VMIs or packages, or alternatively by using 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. In particular, we consider the light-weight virtualization approaches, because, we believe that the combination of light-weight approaches with more standardized software delivery mechanisms can lead to addressing the reproducibility challenge for a wide variety of scientific researchers. A package created by those light-weight 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, PTU [16], chroot, and Docker [3].

We do not claim our solution is the only way to preserve applications. Generally, there are two different approaches to preserve applications: measure the mess and force cleanliness. The first approach allows end users to construct the environment as desired, and then measures the dependencies. The latter one forces users to specify the execution environment for an application in a well organized way. Our objective here is to preserve the mess as-is.

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 in reproducing the applications, and consider the extent to which reproducibility requirements can be satisfied with light-weight virtualization approaches and software delivery mechanisms. We propose an invariant framework for computational reproducibility that combines light-weight virtualization with software delivery mechanisms for efficiently capturing, invariantly preserving and practically deploying applications. We measure the performance overhead of light-weight virtualization and software delivery approaches, and show 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, making it interesting to examine if 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].

Code and data in *TauRoast* is available through five different networked filesystems which are mounted locally, an HDFS cluster for data, 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 Figure 1. Consequently, the real input data may vary depending upon the science question being researched. Similarly the software may name many possible components but the used components are smaller than the named ones.

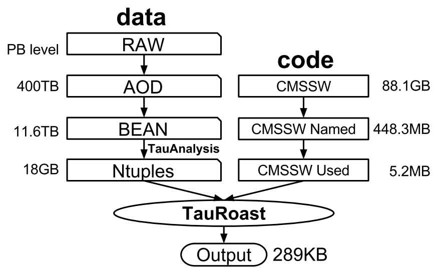

Figure 1: Inputs to Tau Roast

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

3 Challenges in Reproducing HEP Applications

The *TauRoast* and *Athena* application specification was provided to us in the form of an email which described, in prose, how to obtain the source, build the program, and run it correctly on one specific machine at our home institution, with no particular guarantee that it will run anywhere else in the world. This minimal level of documentation about software is routine in the scientific world. We provide the challenges we faced in capturing the application details in a reproducible form and then preserving it for subsequent reuse:

- **Identifying all dependencies.** Due to the distributed nature of HEP applications, these applications depend on a large number of external and local dependencies. External dependencies are often explicitly stated, such as when the application makes connections to Github resources, and CVS servers for downloading source. However, when the application has distributed execution then implicit network connections are present requiring us to identify dependencies on all machines where execution takes place. Implicit local dependencies arise due to 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.** To correctly reproduce an application implies 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 configuration, 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 they 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 much smaller. Often, an entire repository or data source is named within the script, but the program only needs a handful of items from that source. For example, the data is stored on an HDFS filesystem with 11.6TB of data, but only 18GB are actually consumed by the program. Thus there is a size tradeoff 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 get used during program execution.

- **Rapid Changes in Dependencies.** Over the course of three months between collecting the initial email, analyzing the program, and writing this paper, the computing environment continuously changed. 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. In Athena, the computing environment can potentially change daily, since upgrades to the software framework occur on a nightly basis. While the users of this software seem be accustomed to constant change, any preservation technique will have to be very cautious about relying upon an external service, even one that may appear to be highly stable.

- **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 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 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, which if present within large collaborations, 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 be persisted so that they can replayed later without the presence of the commercial software. In effect, a captured application should behave exactly the same way as the application developer intended it to be.

- **Preservation of Captured Entities:** By preservation we imply appropriate mechanisms for (a) documentation of the application development, and (b) automation of any task that becomes inevitably necessary for repeating the application in exactly the same way as the developer executed it.

  Documentation and specification during application development can be an onerous one. The preservation framework must make programming tools available that focus less on documentation, but script, integrate and execute the dependencies so that they are resolved as part of documentation. Automation can extend to various tasks that are 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 barrier to run the application in a new environment.

  In spite of the preservation mechanisms, the application software may not run as intended. For a reproducer's 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 make comparison 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.

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 CVMFS. It works by trapping system calls through the `ptrace` interface, 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 CVMFS. 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.

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 CDE technology 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 kernel versions, application versions, 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.

There are many possible heuristics 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 the greater possibility that it can be adapted to other uses. Other approaches to package generation might including generating the union of multiple dependency sets, or allowing the 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 publically shared repositories. There currently exist a wide variety of services and efforts to share binary objects in this way, and so we assume such multiple services will be available in the future.

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 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 exe-

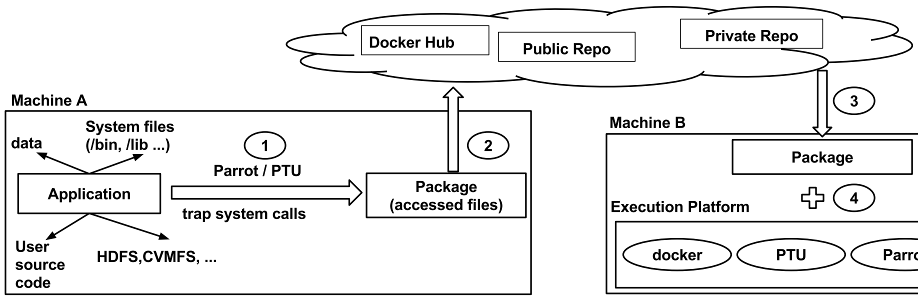

Figure 2: Preservation Framework

Application	Packaging Tool	Obtain Namelist	Create Package	Re-Run (Tool/Time)
Athena	Parrot	10 min 14 sec	00 min 53 sec	Parrot / 09 min 14 sec
Athena	PTU	—	08 min 48 sec	PTU / 07 min 21 sec
TauRoast	Parrot	22 min 50 sec	04 min 25 sec	chroot / 10 min 24 sec
TauRoast	PTU	—	23 min 30 sec	PTU / 08 min 40 sec

Table 1: Time Comparison between Parrot and PTU

cution instances 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 to preserve 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 that contain all the commands to build a Docker image: the computational part is similar to shell scripts and can help provision similar to other provisioning tools (e.g. Chef, Puppet) and the text part is for human consumption and 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 which will check and validate the version of the software being used, and use 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 application using Parrot and PTU. To do this, the application was first executed directly, 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 HDFS, 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 statistics information, and its size is 289 KB. The Athena application processes a given input data file to produce four derived data files. It uses the nightly release of the Athena framework and submits an analysis job to obtain derived data.

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

Table 2: Data and Code Size Used by TauRoast

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; 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. During the package creation procedure, all the input dataset has been copied from HDFS into the package on the local filesystem. So, the re-execution procedure can get its input from the local filesystem instead of obtaining 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, 18GB. 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.

In Athena, the input file is 224MB and output files are 16MB. PTU builds a 855MB package, while Parrot builds an 825MB package. The overhead is of provenance which is 7MB 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.

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.

DOI for the experiment involved in the paper: `doi:10.7274/R0C24TCG`

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 and 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*, pages 11–14. ACM, 2012.

[2] Jakob Blomer, Predrag Buncic, and Thomas Fuhrmann. CernVM-FS: delivering scientific software to globally distributed computing resources. In *Proceedings of the first international workshop on Network-aware data management*, pages 49–56. ACM, 2011.

[3] Carl Boettiger. An introduction to Docker for reproducible research. *ACM SIGOPS Operating Systems Review*, 49(1):71–79, 2015.

[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*, pages 19–22, 2012.

[5] P Buncic, C Aguado Sanchez, J Blomer, L Franco, A Harutyunian, P Mato, and Y Yao. CernVM–a virtual software appliance for LHC applications. In *Journal of Physics: Conference Series*, volume 219, page 042003. IOP Publishing, 2010.

[6] P Calafiura, M Marino, C Leggett, W Lavrijsen, and 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 Student Research Journal*, 2(2):5, 2013.

[8] Serguei Chatrchyan, V Khachatryan, AM 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. *Journal of High Energy Physics*, 2013(5):1–47, 2013.

[9] G Compostella, S Pagan Griso, D Lucchesi, I Sfiligoi, and D Thain. CDF software distribution on the Grid using Parrot. In *Journal of Physics: Conference Series*, volume 219, page 062009. IOP Publishing, 2010.

[10] Philip J Guo and 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, and M Shreeve. Software Preservation Benefits Framework. *Software Sustainability Institute Technical Report*, 2010.

[12] Scott R Kohn, Gary Kumfert, Jeffrey F Painter, and Calvin J Ribbens. Divorcing Language Dependencies from a Scientific Software Library. In *PPSC*, 2001.

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

[14] Tanu Malik, Quan Pham, and Ian T Foster. *SOLE: Towards Descriptive and Interactive Publications.* CRC Press, 2014.

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

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

[17] Quan Pham, Tanu Malik, and Ian T Foster. Auditing and maintaing 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 and Ronald F Boisvert. From scientific software libraries to problem-solving environments. *Computational Science & Engineering, IEEE*, 3(3):44–53, 1996.

[20] Hendrik Schöneberg, Hans-Günter Schmidt, and 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*, pages 359–362. ACM, 2013.

[21] Ivan Subotic, Lukas Rosenthaler, and 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*, pages 29–38. ACM, 2013.

[22] Douglas Thain and Miron Livny. Parrot: An application environment for data-intensive computing. *Scalable Computing: Practice and Experience*, 6(3):9–18, 2005.