IMPROVING THE REPRODUCIBILITY OF SCIENTIFIC APPLICATIONS WITH EXECUTION ENVIRONMENT SPECIFICATIONS

A Dissertation

Submitted to the Graduate School
of the University of Notre Dame
in Partial Fulfillment of the Requirements
for the Degree of

Doctor of Philosophy

by

Haiyan Meng

Douglas Thain, Director

Graduate Program in Computer Science and Engineering
Notre Dame, Indiana
April 2017

IMPROVING THE REPRODUCIBILITY OF SCIENTIFIC APPLICATIONS WITH EXECUTION ENVIRONMENT SPECIFICATIONS

Abstract

by

Haiyan Meng

Reproducibility, a main principle of the scientific method, has historically depended on text and proofs in a publication. However, as computation pervades science and changes the way how research is conducted, relying only on the experimental results in a publication cannot guarantee reproducibility. The execution environment, in which the results were generated, is another important ingredient and must also be preserved to reproduce the results. Unfortunately, execution environments for scientific work are often fragile and too complex to be well understood by researchers, let alone to be preserved.

This dissertation proposes two broad approaches for improving the reproducibility of scientific applications and explore their feasibility and applicability for both single-machine scientific applications and complex scientific workflows. The first approach wraps the minimal execution environment of an application into an all-in-one package. The second approach specifies the execution environment from hardware, kernel and OS all the way up to software, data and environment variables in an organized way, preserves dependencies in the unit of basic OS image, software and data, and combines all the dependencies at runtime using mounting mechanisms.

For each approach, a prototype was implemented and the following three aspects are explored: what to preserve, how to preserve and how to reproduce. The time

and space overheads to preserve and reproduce applications, and the correctness of preserved artifacts are evaluated through applications from high energy physics, bioinformatics, epidemiology and scene rendering. The evaluation results show that both approaches allow researchers to reproduce an application and verify its results. However, the second approach avoids storing shared dependencies repeatedly and makes it easier to extend the original work.

This work makes its contribution by demonstrating the importance of execution environments for the reproducibility of scientific applications and differentiating execution environment specifications, which should be lightweight, persistent and deployable, from various tools used to create execution environments, which may experience frequent changes due to technological evolution. It proposes two preservation approaches and prototypes for the purposes of both result verification and research extension, and provides recommendations on how to build reproducible scientific applications from the start.

CONTENTS

FIGURES

TABLES

ACKNOWLEDGMENTS

This work would not have been possible without the guidance and encouragement of my advisor, Dr. Douglas Thain. His positiveness, ingenuity, impressive teaching methods and wonderful class design made him a great role model for me. I would also like to thank my dissertation committee members, Dr. Jaroslaw Nabrzyski, Dr. Michael Hildreth and Dr. Sandra Gesing, for their help and suggestions for my work.

Thanks my fellow grad students from the Cooperative Computing Lab for all the interesting discussions. Thanks our collaborators from the Department of Physics for providing interesting use cases for my research and always giving lots of helpful feedback. Thanks M.D. McNally from the Departement of Computer Science and Engineering and the staff from the Center for Research Computing for providing critical technical assistance throughout this research effort.

Special thanks to my fellow lab member and my now-fiancé, Dr. Patrick Donnelly, for teaching me how to find not just the solution, but the correct one, how to be patient, and how to be a true Fighting Irish fan.

I am grateful for the financial support provided by National Science Foundation grants PHY-1247316 (DASPOS), OCI-1148330 (SI2) and PHY-1312842. Without this support, I would not have been able to focus on my research.

Thanks to Joyce Yeats, Dian Wordinger and Ginny Watterson for their prompt and kind help for all administrative affairs.

Last but not least, thanks to my family for supporting my pursuing of a Ph.D. in the USA. Thanks to my mom, Xinping Liu, for always having the deepest trust and support on my decisions. Thanks to my dad, Jianbin Meng, for teaching me how

to do things with high standards. Thanks to my sister, Ge Meng, for always being my sweetheart and angel. Thanks to my brother, Peng Meng, for taking care of our parents while I am far away from home.

CHAPTER 1

INTRODUCTION

Reproducibility is one main principle of the scientific method [22, 87]. It advances science by first making it possible to verify a scientific result, and then increasing the chance of reusing the result or extending the work. Ensuring the reproducibility of a scientific result, however, often entails detailed documentation and specification of the involved scientific work.

Historically, text and proofs in a publication have achieved this end. However, as computation pervades science and changes the way how research is conducted, relying only on text and proofs in a publication is no longer sufficient [100]. Experimental results may be plotted into beautiful figures and presented in an academic conference or published in a journal, however, descriptions of the actual procedures by which the results were achieved are often superficial and imprecise. Authors may mention the platform configuration used for their experiments in the Evaluation section, such as CPU, memory, network and disk, but seldom include the details of the software stack, such as software version, dataset source and analysis scripts.

Without a complete description of the execution environment used by the original authors, it may be difficult or even impossible to reproduce scientific work. For example, a study in 2011 found that only 6% medical studies in pharmacology were completely reproducible [91], and another study in 2012 of 53 medical research papers in cancer research found that 47 of them were irreproducible [16]. A study in 2015 of the repeatability in computer systems research examined 402 papers from ACM conferences and journals whose results were backed by code, and found that only

85 of the papers provided links to their code in the paper itself. The study also showed that only the code in 32.3% of the papers can be rebuilt within 30 minutes, the code in another 16% of the papers can be rebuilt with extra effort of debugging, and it was difficult to rebuild the code in the remaining 51.7% of the papers [92]. Therefore, apart from the experimental results published in journals or conference proceedings, the execution environment, in which the results were generated, must also be preserved to reproduce the results.

Unfortunately, execution environments for scientific applications are fragile due to the rapid changes of explicit and implicit dependencies. The kernel and OS (Operating System) may upgrade, the software may have a new version, the data format may change [61], the network dependencies may be unavailable due to linkrot [109], or some core staff may leave. In addition, the hardware architecture an application depends on today may become obsolete five or ten years later.

Execution environments for scientific applications are getting more complex in both the variety and size of dependencies [76, 97]. Consider a HEP (High Energy Physics) application [75] running on a single machine: four different networked filesystems which are mounted locally are used to deliver software and data dependencies - HDFS [23], CVMFS [19], PanFS [113] and AFS [98]. In addition, some software dependencies are delivered through CVS (Concurrent Versions System) [29], the analysis code is delivered through Git [69], some libraries are retrieved from a third-party website, and a header file is retrieved from a personal website. The complete size of its input dataset is a staggering 11.6TB and the size of the software framework used is 88.1GB, often undeployable on a single system.

Execution environments for scientific workflows [103], such as Pegasus [37], Makeflow [11], Taverna [83] and Swift [115], introduce new complexity in that multiple software stacks are often involved and the communication between multiple execution nodes need to be considered. Ideally, scientific workflows should improve the

reproducibility of scientific applications by making it easier to share and reuse workflows between scientists. However, scientists often find it difficult to reuse others' workflows, which is known as *workflow decay* [53]. For example, myExperiment [48] is designed to be a sharing social website which allows scientists to share their workflows. However, a study of Taverna workflows on myExperiment shows that 80% of the workflows on the site could not be reproduced, mainly due to the volatility of the resources required for workflow executions [119].

Documenting execution environments is often not given high priority by researchers due to the time pressure from publication deadlines and moving on to the next challenges in their research. In the case where they are willing to preserve the execution environments, the complicated dependencies of scientific applications often stop them. What's more, even system administrators are often frustrated by the dependency hell problem [51].

Various attempts have been made to enhance traditional scholarly publication and improve the reproducibility of scientific results, such as *Research Objects* [15], dynamic documents [46, 82], reproducible papers [45], virtual machines [65] and virtual appliances [57]. However, these solutions may only bundle together the necessary resources and are not directly executable; or not cover the complete execution environment (e.g., the hardware, kernel and OS dependencies are sometimes missing); or fail when the software stack is very large [76]; or not be space-efficient because common dependencies shared by multiple scientific applications such as shared libraries are preserved redundantly.

The existing efforts to improving the reproducibility of scientific workflows are mainly guidelines on how to design reproducible scientific workflows [15, 47, 53, 53]. These guidelines focus on the specifications of scientific workflows and ignore the other components of scientific workflow systems [116], which usually include multiple layers - the workflow specification layer, the specification parser layer, the task scheduler

layer and the computing resource layer. In addition, the users of scientific workflow systems often do not have control over the underlying execution environments used to run their tasks, and sysadmins must respond to the cumbersome job of configuring the execution environments on all the execution nodes to meet the requirements of different workflows. Otherwise, the tasks would fail on the nodes with incompatible execution environments, leaving these execution nodes not fully exploited.

This dissertation proposes two broad approaches for facilitating the reproducibility of scientific applications and explore their feasibility and applicability for single-machine scientific applications and complex scientific workflows. The first approach wraps the minimal execution environment of an application into an all-in-one package that can be used to reproduce the application. The second approach specifies the execution environment from hardware, kernel and OS all the way up to software, data and environment variables in an organized way. This enables preserving dependencies in basic units of OS image, software and data. All the dependencies of an application are combined into a unified sandbox at runtime using mounting mechanisms. A prototype was implemented for each approach - *Parrot Packaging Tool* for the first approach and *Umbrella* for the second.

For each approach, the following three aspects are explored: what to preserve, how to preserve, and how to reproduce. The time and space overheads to preserve and reproduce applications, and the correctness of preserved artifacts are evaluated through applications from high energy physics, bioinformatics, epidemiology and scene rendering. The evaluation results show that both approaches allow researchers to reproduce an application and verify its results with acceptable effort. However, the second approach avoids storing shared dependencies repeatedly and makes it easier to extend the original work.

This work makes its contribution by demonstrating the importance of execution environments for the reproducibility of scientific applications, and differentiat-

ing execution environment specifications, which should be lightweight, persistent and deployable, from various tools used to create execution environments, which may experience frequent changes due to technological evolution. It proposes two preservation approaches and provides two prototypes to assist researchers in reproducing their applications for the purposes of both result verification and research extension. Finally, this work provides recommendations on how to build reproducible scientific applications from the start.

1.1 Problem Scope and Domain

The definition of reproducibility is still under debate, and the key contention lies in whether the reproducing procedure uses the original artifacts or not [43]. Instead of arguing its definition, *this work focuses on the feasibility evaluation of different preservation methods and the cost comparison of preserving and reproducing scientific applications using different preservation methods.*

The preservation objectives of reproducible research can be classified into two categories: **result verification** and **research extension**. Result verification can be further divided into identical environment verification, new environment verification and new software implementation verification. Research extension can be divided into new data extension and new software extension [106]. Result verification helps build up the trust for existing research results, while research extension advances science by allowing researchers to explore new ideas. *This work aims to assist researchers in both the result verification and research extension aspects of reproducible research.*

Verification of scientific results has two different targets: **output-oriented** and **performance-oriented** [96]. Performance verification can be very challenging or even impossible due to the impact of fast hardware upgrade cycle [52] and runtime factors, such as network quality, disk speed, CPU load and memory size. Sometimes even output verification is difficult because either the application is stochastic (e.g.,

Monte-Carlo simulation in the CMS experiment [35]) or the usage of complex high performance computing systems introduces non-determinism [31, 72]. *This work focuses on the case where the outputs of different runs of a scientific application are identical literally.*

The target of preservation may be a **problem**, a **solution** or an **implementation**. A scientific application is an implementation of a specific solution to a given problem. For example, A problem may be sorting 100,000 numbers, quicksort is one solution for the problem, and one implementation of quicksort may be a C program which is supposed to run on a x86_64 machine. *A preservation plan should cover the problem, the solution and the implementation.* Preserving the implementation allows others to verify the results easily, utilize the implementation to sort other data sets, or even optimize the implementation. Preserving the solution makes it possible for new implementations, which can be used to compare with the preserved implementation for the purpose of verification. Preserving the problem itself makes it possible for new solutions, which can be used to verify the correctness of existing solutions, and may have better performance.

Computing and Computing Resources. In general, a program (i.e., computing) is not tightly coupled with any single server (i.e., computing resources). For example, the C quicksort program mentioned above can run on any server which satisfies the dependency requirements. Aware of the difference between computing and computing resources helps determine the preservation scheme.

Software Framework and its Deployments. It is also important to differentiate a software framework and its different deployments. For example, HTCondor [108], as a software framework, can be deployed in different universities or organizations. A scientific result may be obtained through an experiment running on the HTCondor pool at the University of Notre Dame. However, the preservation specification should not be tightly coupled with the HTCondor deployment, instead

should allow the scientific result to be reproduced on other HTCondor deployments or other software frameworks such as Apache Hadoop [105].

Software/Data and Software/Data Delivery Tools. Another difference to be aware of is between software/data and software/data delivery tools. For example, CMSSW (CMS SoftWare) is the software framework for analyzing CMS data [61], and currently can be delivered to end-users through a virtual file system called CVMFS (Cern Virtual Machine File System) [19]. CMSSW is crucial for the execution of CMS applications, however, CVMFS, the common way today, is not the only way to delivery CMSSW. On a server with enough storage space, it is possible to leave CVMFS alone and directly put CMSSW onto local disks. Therefore, the preservation of CMSSW and CVMFS should be considered separately. Another good example is the petabyte-level CMS data and the XRootD file access system, which can be used to assist concurrent access to the CMS data from multiple clients [41].

1.2 Motivation

The work is highly motivated by the Data And Software Preservation for Open Science (DASPOS) project [71]. DASPOS aims to explore the key technical problems involved in preserving and reproducing high energy physics experiments, which already accumulated petabyte-level data sets and keep generating more data, and use complex and evolving software frameworks to analyze these data sets.

This work began with a thorough case study of a high energy physics experiment, called *TauRoast*, conducted by the high energy physics researchers from the Department of Physics at the University of Notre Dame. The case study revealed the variety and size of the software and data dependencies of *TauRoast*, and the complexity of execution environment configurations for high energy physics applications [75].

Several refinements were done to improve the reproducibility of the *TauRoast* experiment. Different preservation approaches were explored to improve the repro-

ducibility of existing scientific applications. The lessons learned throughout the exploration may guide the design of reproducible scientific applications from the start.

1.3 Contribution

This work aims to facilitate the reproducibility of scientific applications including both complex single-machine scientific applications and scientific workflows. It makes several contributions in this regard:

First, it explores the challenges in preserving scientific applications, and evaluates the reproducibility of different components in complex execution environments.

Second, it proposes two broad approaches for conducting reproducible research. The first approach preserves the minimal execution environment into an all-in-one package in a messy way. The second approach preserves the dependencies in the unit of OS image, software and data, specifies an execution environment from hardware, kernel and OS all the way up to software, data and environment variables in an organized way, and brings all the dependencies into a unified sandbox during runtime through mounting mechanisms. The pros and cons of each approach are explored, which helps researchers choose proper preservation plans according to their preservation aims.

Third, it separates the preserved artifacts, the preservation tools which create the preserved artifacts, and the reconstruction tools which recreate the execution environment with the preserved artifacts and reproduce the original work. This improves the portability of the preserved artifacts and allows new technologies to reuse the preserved artifacts.

Fourth, it summarizes the characteristics of reproducible scientific applications, and gives recommendations on how to design reproducible scientific applications from the start and how to build tools for conducting reproducible research.

1.4 Publications

- *A Case Study in Preserving a High Energy Physics Application with Parrot* in the Journal of Physics: Conference Series (JPCS) [75]. This paper describes the complexity of preserving a complex high energy physics application, *TauRoast*, and proposes a prototype, *Parrot Packaging Tool*, which can be used to capture the minimal execution environment of a scientific application into an all-in-one package. The preserved package can be re-executed using a variety of technologies including Parrot [107], Docker [80], chroot [64] and virtual machines [14].

- *An Invariant Framework for Conducting Reproducible Computational Science* in the Journal of Computational Science (JOCS) [76]. This paper proposes a preservation framework for computational reproducibility, which covers: how to use system call trapping, a light-weight virtualization technique, to create a reduced package which only includes all the necessary dependencies; how to distribute preserved artifacts through standard software delivery mechanisms like Docker; and how to recreate applications with the help of the preserved artifacts through flexible deployment mechanisms such as Parrot, PTU [88], Docker and chroot.

- *Umbrella: A Portable Environment Creator for Reproducible Computing on Clusters, Clouds, and Grids* at the 2015 Workshop on Virtualization Technologies in Distributed Computing (VTDC) [77]. This paper presents *Umbrella*, a prototype for specifying and materializing execution environments in a portable and reproducible way. Umbrella accepts a declarative execution environment specification for an application, and then determines the minimum technology needed to deploy it. The application may be run natively if the local execution environment is compatible, or run inside a container, a virtual machine, or remotely in the cloud.

- *Techniques for Preserving Scientific Software Executions: Preserve the Mess or Encourage Cleanliness?* at the 2015 International Conference on Digital Preservation (iPres) [106]. This paper explores from a high level what techniques and technologies must be put in place to allow for the accurate preservation and reproduction of scientific software, discusses the fundamental problems of managing implicit dependencies, and outlines two broad approaches to preservation scientific software: preserving the mess and encouraging cleanliness.

- *Conducting Reproducible Research with Umbrella: Tracking, Creating, and Preserving Execution Environments* at the 2016 IEEE International Conference on eScience [79]. This paper proposes a framework facilitating the conduct of reproducible research by tracking, creating and preserving the comprehensive execution environments with Umbrella. The framework includes a lightweight,

persistent and deployable execution environment specification, an execution engine which creates the specified execution environments, and an archiver which archives an execution environment into persistent storage services like Amazon S3 and OSF (Open Science Framework). The execution engine utilizes sandbox techniques like virtual machines, Linux containers and user-space tracers, to create an execution environment, and allows common dependencies like base OS images to be shared by sandboxes for different applications.

- *Facilitating the Reproducibility of Scientific Workflows with Execution Environment Specifications* at the 2017 International Conference on Computational Science (ICCS) [78]. This paper explores the challenges in reproducing scientific workflows, and proposes a framework for facilitating the reproducibility of scientific workflows by giving scientists complete control over the execution environments for their workflows and integrating execution environment specifications into scientific workflow systems. The framework allows the execution environment dependencies to be archived separately in finer granularity instead of gigantic all-in-one images. A prototype is implemented by integrating *Umbrella* [79], an execution environment creator, into *Makeflow* [11], a scientific workflow system.

1.5 Overview of Dissertation

This chapter gives an introduction to this work, clarifies the problem scope and domain this work aims to solve, explains the motivation and contribution of this work, and lists the relevant publications.

CHAPTER 2: RELATED WORK reviews several topics directly related to this work and explain the research gap this work aims to fill: the three possible approaches to preserving scientific applications; different types of virtualization techniques and how they can facilitate the conduct of reproducible research; single-machine scientific applications and scientific workflows; the ways of configuring complex execution environments for scientific applications; the reproducibility in clouds, grids, and HPC systems; the management and distribution of scientific data and software; and data provenance.

CHAPTER 3: REPRODUCING SINGLE-MACHINE SCIENTIFIC APPLICATIONS WITH MINIMAL EXECUTION ENVIRONMENT

PACKAGES describes a case study in reproducing a high energy physics application; summarizes the challenges in reproducing complex scientific applications learned through the case study; explains the several refinements to make scientific applications reproducible; describes and evaluates a prototype, *Parrot Packaging Tool*, which can be used to create minimal execution environment packages and reproduce complex scientific applications.

CHAPTER 4: REPRODUCING SINGLE-MACHINE SCIENTIFIC APPLICATIONS WITH EXECUTION ENVIRONMENT SPECIFICATIONS explains the complexity of execution environments for scientific applications which makes it difficult to reproduce experiment results; describes a framework which aims to help researchers track, create and preserve their execution environments; provides two example workflows illustrating how the framework can facilitate the reproducibility of scientific applications; and evaluates the correctness of the framework, the time and space overheads of preserving and creating execution environments.

CHAPTER 5: REPRODUCING SCIENTIFIC WORKFLOWS WITH UMBRELLA lists the challenges in reproducing scientific workflows; describes a framework facilitating the reproducibility of scientific workflows by giving scientists complete control over the execution environments for their workflows and integrating execution environment specifications into scientific workflow systems; describes a prototype of the framework by integrating *Umbrella*, an execution environment creator, into *Makeflow*, a scientific workflow system which can utilize computing resources from a HTCondor cluster; and presents the evaluation results of the framework on its correctness, time and space overheads to preserving and creating execution environments.

CHAPTER 6: CONCLUSION summarizes this work; explains how this work fills up some, not all of the research gap in reproducible research, and improves the reproducibility of scientific applications in terms of both result verification and research

extension; summarizes the characteristics of reproducible scientific applications; and gives some recommendations on how to build reproducible scientific applications from the start to researchers and on how to build tools for facilitating the reproducibility of scientific applications to tool creators.

1.6 A Note on Terms

Single-Machine Scientific Applications in this work means applications whose computing and storage needs can be satisfied by the resources provided on a single machine. Applications involving accesses to remote filesystems, which can be mounted locally and accessed via local file paths like `/afs/nd.edu/user27/harry`, are still counted as single-machine applications.

Machine in this work means commonly used workstation or server, and does not mean supercomputer.

Reproducibility in this work means the ability of both repeating an application exactly and extending the original application.

Preserved Artifacts in this work means the materials relative to an application being preserved, which makes it possible to reproduce the application. Some examples are OS images, software, data sets, and documentations about its execution environment, background, motivation and goal.

Here is a list of terms which may be used alternatively according to the context:

- **scientific applications**, **applications**, and **experiments**;
- **machine** and **server**;
- **re-use**, **re-run**, **re-execute**, **repeat**, and **reproduce**;
- **analysis code** and **analysis script**;
- **researchers** and **scientists**;
- **scientific results**, **research results** and **experiment results**.

CHAPTER 2

RELATED WORK

This chapter reviews several topics directly related to this work and explains the research gap this work aims to fill. Section 2.1 outlines the three possible approaches to preserving scientific applications. Section 2.2 discusses different types of virtualization techniques and how they can facilitate the conduct of reproducible research. The methods of configuring complex execution environments for scientific applications are reviewed in Section 2.3. Section 2.4 reviews scientific workflows, scientific workflow systems, workflow decay and different degrees of control over the underlying execution environments. The reproducibility in clouds, grids and HPC systems is reviewed in Section 2.5. Section 2.6 gives an overview of the management and distribution of scientific data and software. Section 2.7 discusses the topics of data provenance.

2.1 Preservation Approaches of Scientific Applications

Scientists implement their solutions to complex scientific problems as *scientific applications* with the help of computational science [58, 93]. The successful execution of a scientific application depends on the correctness of the solution, the computer software implementing the solution, and the underlying computer hardware such as CPUs, memory, disk and networking devices.

To facilitate the reproducibility of computational science, there are three main strategies to preserve scientific applications: hardware preservation, migration, and emulation [74, 118].

Hardware preservation maintains the original execution environment for an application including both the hardware and software. The reproduction is easy by re-running the same application on the same execution environment. However, this approach is not very efficient due to the cost and space overheads of preserving both the software and hardware, and may not be feasible sometimes, especially for distributed applications which involve multiple computing nodes. The reproduction may be impossible from a long-term perspective because the hardware may be damaged by factors like humidity or the lifetime of disks may end.

Migration adapts old software to future platforms by modifying or even rewriting the software. The old versions of software and its dependencies do not need to preserved. However, this approach is only feasible when the software is still under development and well maintained.

Emulation keeps the software unchanged and recreates the original execution environment on the future platforms through virtualization techniques [14]. The approach is cost-efficient because the reconstructed execution environment can be shared and re-used by different applications. It also moves cost of porting many scientific applications to maintaining a single emulation framework.

Each of these approaches has its pros and cons, and may be especially useful or even demanded for certain circumstances or research fields. *This work mainly focuses on the emulation approach.*

2.2 Virtualization and Reproducible Research

Virtualization techniques can be used to virtualize computer hardware platforms, operating systems, applications, memory, storage, networks and so on. Research, including this work, has been exploring on how to utilize different types of virtualization techniques to recreate the original execution environments of scientific applications and conduct reproducible research.

2.2.1 Hardware Virtualization

In hardware virtualization, hypervisors (i.e., virtual machine monitors) such as VirtualBox [112] and VMware ESX [111] can be utilized to create a VM (Virtual Machine) on a host machine. A virtual machine emulates a computer system with an operating system. A hypervisor manages the execution of virtual machines on the host machine. Multiple instances of different operating systems may run on the same host machine. For example, a physical x86 machine may run Microsoft Windows, Linux and Mac OS X concurrently.

The capability of running another OS, which may be different from the OS on the host machine, makes hardware virtualization useful for researchers to share their work conveniently. The original author can wrap the whole execution environment for his or her experiment into a VMI (Virtual Machine Image), then distribute and share the VMI [24, 62]. Other researchers can reproduce the original experiment with the VMI on their servers if the required hypervisor is available. Cloud computing platforms, such as the Amazon Elastic Computing Cloud [62] and Google Compute Engine [63], make it possible to reproduce an experiment on cloud resources, without the necessity of installing hypervisors on local machines [57].

This approach does not work for complex scientific applications with large software stacks sizing terabytes (TBs), such as some high energy physics applications [75]. A virtual machine image including all the required dependencies can be used to repeat an experiment accurately, however, the structure of the VMI is often complicated and not well-documented, making it very difficult for others to understand the configurations of the experiment and to extend the original work. Hardware virtualization also involves execution overheads because each instruction within a VM needs to be translated into an instruction on the host machine via the hypervisor.

2.2.2 Operating System-Level Virtualization

OS-level virtualization allows multiple OS instances to run simultaneously on top of a single OS kernel [117]. Compared with hardware virtualization, this has lower execution overhead because no hardware-level instruction translation is needed. With OS-level virtualization, only OS instances supported by the OS kernel on the host machine can run concurrently, such as different Linux distributions can run on top of the Ubuntu Linux distribution. There is no way to run Microsoft Windows and Linux simultaneously with OS-level virtualization. In spite of its low-overhead and flexibility, the security of OS-level virtualization has been a concern [94]. The implementation of OS-level virtualization may only deploy file system isolation (i.e., mount namespace isolation), such as chroot [64]; or isolate file system, network, process, user namespace and so on, such as Docker [81].

OS-level virtualization, such as Docker, has been used to facilitate the reproducibility of computational science [20]. Using Docker, researchers can share their work in two ways. The first one is to create a Docker image including a whole execution environment and then share the image. The second one is to create a *Dockerfile*, which is a recipe specifying all the steps of configuring an execution environment and can be used to build a Docker image, and share the *Dockerfile*. However, building a *Dockerfile* at different times may result in different images due to the updates to recipe components such as base Docker image and remote network dependencies.

2.2.3 Application Virtualization

To decrease the space overhead of preserved artifacts, application virtualization techniques, such as CDE [51], PTU [89] and ReproZip [32], trap the system calls of an application and only preserve the actually used files. However, like VMI, shared library dependencies are repeatedly preserved multiple times. The context and structure of preserved artifacts are not clear, which makes it difficult to reuse. In addition,

all these approaches mainly focus on the preservation of local dependencies, and do not solve the problem of how to preserve network dependencies. They preserve an application as it is and do not give any guidance or recommendations on how to design scientific applications in a reproducible way from the start. *This work provides solutions for tracking local and network dependencies, and gives recommendations on how to build reproducible scientific applications from scratch.*

2.3 Execution Environment Configuration Approaches

The requirement of large number of computing resources for accelerating the execution of scientific applications demands an efficient way to configure execution environments on possible heterogeneous execution nodes, especially for grid computing. One approach to deploy execution environments on a large number of execuiton nodes is disk cloning [60], which copies the contents of a computer hard disk to another disk directly or with the help of a disk image file.

Software configuration management systems [38] like Puppet [70], Chef [104] and CFEngine [95], allow system administrators to specify the desired configuration for each managed device without considering the heterogeneity and complexity of the devices. This increases the scalability of system configuration by avoiding repetitive tasks and reduces errors introduced by manual configuration through automation. However, software configuration systems do not consider the key problems of reproducible research, like data provenance, context description and access right [102].

Execution environment management frameworks, such as Grid'5000 [21], FutureGrid [110] and VMPlants [65], aim to ease the execution environment configuration for grid computing. FutureGrid even allows an experiment to be reproducible through recording the user and system actions. However, the environment construction in these systems are accomplished either by a virtual appliance or by the combination of a base virtual machine image and a series of configuration steps. Delivering a whole

software stack through virtual machine images is expensive and makes it difficult to reuse shared dependencies.

2.4 Scientific Workflows and Scientific Workflow Systems

According to the size and complexity, scientific applications can be split into two categories: single-machine scientific applications and scientific workflows. A single-machine scientific application can meet its computing and storage requirements with the resources on a single machine. When an application is too big to be solved on a single machine, scientific workflows [103] can be used to disseminate complex data transformations and analysis procedures into an ordered list of smaller and possibly independent tasks, which allows computing resources from clusters [26], grids [17] and clouds [13] to be utilized to accelerate the pace of scientific progress.

To make it easy for scientists to compose and execute scientific workflows, a variety of *scientific workflow systems* have been developed [116], such as Taverna [83], Swift [115], Pegasus [37] and Makeflow [11]. The end-users of these workflow systems only need to specify an ordered list of tasks. The workflow systems respond to communicate with execution engines, schedule tasks to the underlying computing resources, manage data sets and deliver fault-tolerance.

Ideally, scientific workflows should improve the reproducibility of scientific applications by making it easier to share workflows between scientists. However, scientists often find it difficult to reuse others' workflows, which is known as *workflow decay* [53]. For example, myExperiment [48] is designed to be a sharing social website which allows scientists to share their workflows, however, a study of Taverna workflows on myExperiment shows that 80% of the workflows on the site could not be reproduced, mainly due to the volatility of the resources required for workflow executions [119].

Guidelines on how to design reproducible scientific workflows were proposed in [47, 53], and *Research Objects* [15] were used to provide detailed metadata information

about workflows [53]. However, these guidelines focus on the specification layer of scientific workflow systems and ignore the other components of scientific workflow systems, such as the specification parser layer, the task scheduler layer, and the computing resource layer.

Depending on the scientific workflow systems used, scientists have different levels of control over the underlying execution environments. Pegasus [37] and Swift [115] allow scientists to compose *abstract workflows* without worrying about the details of underlying execution environments, which means sysadmins must respond to the cumbersome job of configuring computing resources to meet all the requirements of different workflows. Makeflow [11] allows executables to be specified in workflow specifications and delivered to execution nodes at runtime. This is simple but not always correct, because executables may be sent to execution nodes with incompatible execution environments. To fix this, Makeflow allows scientists to specify a gigantic all-in-one image like a Docker image [80] containing the required execution environment and delivers the image to execution nodes at runtime [120]. This gives scientists more control over execution environments, but introduces gigantic images to preserve and store.

This work gives scientists complete control over the execution environments for their workflows by integrating execution environment specifications into scientific workflow systems. It also allows the execution environment dependencies to be archived separately in finer granularity instead of gigantic all-in-one images. Archiving dependencies in finer granularity allows the dependencies to be updated in different paces.

2.5 Reproducibility in Clouds, Grids and HPC Sytems

Computing and storage resources from clouds, grids and HPC (High-Performance Computing) systems accelerate the pace of scientific progress, however, may make scientific results less reproducible [84].

With cloud computing, researchers run their experiments on the resources provided by cloud providers, and only have limited control over the cloud platform's technology [67]. For example, the Amazon EC2 updates supported instance types periodically, which may cause some instance types to be unavailable permanently. Storing research data in clouds causes concerns about security, privacy, and costs of both accessing and storing data [85]. In addition, an AMI (Amazon Machine Image) is not globally unique but only unique within a region; copying an AMI to another region results in an AMI with its own unique identifier [4].

The resources in grids are usually heterogeneous in terms of hardware architecture, kernel version, OS type, CPU number, memory size and disk size [77]. A scientific application tuned perfectly on a Debian machine may fail to execute on an Ubuntu machine due to the incompatibility of execution environments.

High-end HPC systems have been widely used in modern computational simulation [66]. However, its internal complexity, operation order uncertainty and floating-point arithmetic introduce non-determinism and conflict with the goal of making scientific research reproducible [31, 40].

2.6 Management and Distribution of Data and Software

As the size of scientific data increases, data management has become more and more important [50]. The data grid [30] was proposed to manage scientific data and its metadata. The XRootD file access system [41] was used to assist the concurrent access to PB-scale data sets from multiple clients.

Compared with the TB-scale or even PB-scale size of scientific data, software used to analyze scientific data is much smaller - MB-scale or GB-scale. However, there are often multiple versions or releases of software. For example, CMSSW, the software collection used for CMS experiments, has dozens of releases spanning from CMSSW4_1_X to CMSSW9_0_X [5]. Version control systems like Git [69] and

CVS [29], were designed to track the change history of documents and codes. CVMFS was designed to deliver scientific software such as CMSSW to globally distributed computing resources [19].

When the reproducibility of scientific applications is considered, it is important to differentiate scientific data and software from the software frameworks used to deliver and access scientific data and software. The latter may suffer frequent changes as new data access and transfer protocols become available.

DOI (Digital Object Identifier) provides a persistent identifier to uniquely identify an object, and can be used to identify scientific data and publications [86]. However, the DOI system recommends, but does not force, the objects referred by a DOI name to be persistent or immutable. For example, the DOI name 10.1000/182 always refers to the latest version of the DOI handbook, which is the primary source of information about the DOI system. A DOI often points to an overview page including the information about the object and links to auxiliary resources. To guarantee data integrity, checksum algorithms such as MD5 and SHA-1 should be used.

2.7 Data Provenance

Data provenance facilitates reproducibility by making it possible to judge the quality of the data, track back sources of errors and give credit to original data creators. The support for data provenance in different systems varies widely.

Data provenance in a database was explored and a query-rewriting framework which guarantees both where-provenance and why-provenance was proposed in [25]. A data provenance architecture to deal with the data provenance in web and grid services was proposed in [101], which records data provenance by recording all web service invocations carried out by a client, stores data provenance information in a relational database and supports for provenance browsing and validation through a query interface.

CDE [51] records and preserves the local dependencies, but does nothing to preserve the third-party network dependencies. Reproducing an experiment with the preserved artifacts would fail if any network dependency is unavailable. PTU [88] traps the network-relevant system calls and records all the recognizable network data into its own database, which supports replaying the network access offine. However, the model fails if the network data is encrypted through SSL [54] or TLS [39]. Kameleon [97] utilizes a caching web proxy to cache all the data coming from the network and guarantees reproducing an experiment always refers to the network resources from the caching web proxy. However, special considerations are necessary to deal with proprietary data and huge data dependencies. In addition, similar to PTU, the web proxy can not cache encrypted data. Although there exists researches on how to intercept encrypted data in a SSL/TLS channel [27, 36, 59], these researches face challenges such as the legal problem involved and the huge computing resources required.

CHAPTER 3

REPRODUCING SINGLE-MACHINE SCIENTIFIC APPLICATIONS WITH MINIMAL EXECUTION ENVIRONMENT PACKAGES

3.1 Introduction

Reproducibility is an important principle of computational science [22]. 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 science 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 [73].

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 single-machine scientific application involving a modest amount of computation and data, this could be accomplished by capturing the complete environment, data, and software within a single VMI (Virtual Machine Image) [18, 49],

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 [28].

However, this simple approach may not always be 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 a particular data analysis package is needed for an application, 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 [34], or a scalable filesystem [19]. For such resources, it must be decided whether the dependency will simply be noted, or if it must be incorporated whole or in part.

This chapter presents a case study of reproducing one complex high energy physics application, *TauRoast*, which was shared first in the form of an email that describes in prose how to install the software and run the analysis. The background information of *TauRoast* and its code and data sources, are provided in Section 3.2. The challenges involved in reproducing complex scientific applications are illustrated in Section 3.3. Section 3.4 explains the three-stage refinements to convert the original email into an executable and preservable object. A prototype, called *Parrot Packaging Tool*, is presented in Section 3.5, which aims to assist in the measurement and preservation of implicit dependencies for complex applications, and reduce the size of the dependencies that are necessary for a preserved object to function. Section 3.6

explores the methods to archive, distribute and deploy the preserved pacakges. Finally, section 3.7 demonstrates the preserved object, created using *Parrot Packaging Tool*, functions correctly in three different physical and cloud environments.

3.2 A Case Study in Reproducing A High Energy Physics Application

TauRoast, the high energy physics application used in the case study, 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.

3.2.1 Code Sources of the Application Under Study

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. These software dependencies vary widely in their standardization degrees: 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 framework known as CMSSW (CMS SoftWare) [33], 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 CVS (Concurrent Versions System), or by installing a complete binary package on a shared filesystem within a HPC (High

Performance Computing) center. In recent years, distribution has moved to an on-demand delivery system known as CVMFS (Cern Virtual Machine File System) [19], 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.

3.2.2 Data Sources of the Application Under Study

The CMS collaboration provides end users with a pre-processed and reduced data format, AOD (Analysis Object Data) [55], 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 worldwide, 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 the University of 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 is reduced to the ROOT files [12], which contain 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.

There are totally 9 different data and code dependencies, as shown in Table 3.1. The *Total* column shows the total size of each dependency, the *Named* column shows the size of the part of each dependency mentioned in the analysis script, and the *Used* column shows the actual used size of each dependency.

Figure 3.1 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.

3.3 Challenges in Reproducing Complex Scientific Applications

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 the University of Notre Dame, with no guarantee that it would work successfully on a new machine. Although this starting point may seem extreme,

TABLE 3.1

DATA AND CODE USED BY TAUROAST

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

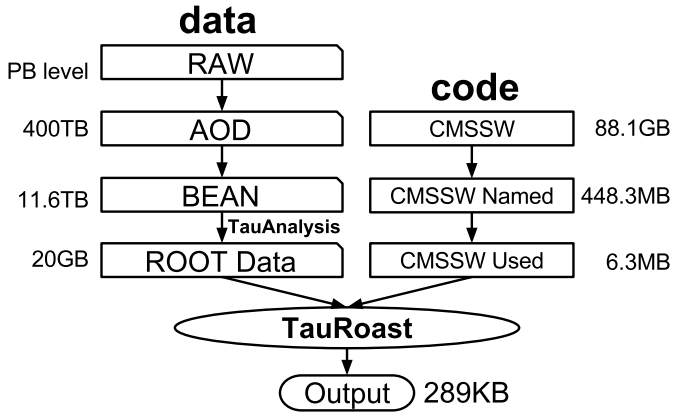

Figure 3.1. Code/Data Inputs to TauRoast and Code/Data Sizes

it is natural for collaborators to share configurations in this form, and to rely on the presence of a working environment already installed.

To prepare the application for permanent archival, we first elaborated the email instructions into an executable script that declares the necessary environment variables, downloads and checks out source codes from several different locations, builds the codes 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 of complex single-machine scientific applications:

- *Many Explicit External Dependencies. TauRoast* depends on a large number of external dependencies, each with a different access method and data source, as shown in Table 3.1. While it was knew in advance that it depended upon the large CMSSW framework, it was not apparent until elaborating the script that it depended upon two different Github repositories [9, 10] for the Tau source, a CVS server at CERN for some configuration information, a public website for the PyYAML library [3], and the public home page of a Notre Dame student for one missing header file. 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 [23] cluster, some configuration data was stored on a CVMFS [19] filesystem, and a variety of software tools were on NFS [56], PanFS [113], and AFS [98] systems. All these networked filesystems were installed and provisioned by local system administrators, the original author may not be aware of all of them.

- *Configuration Complexity.* As a means of controlling the complexity of dependent software packages, the high energy physics community has developed a number of tools that perform run-time configuration and consistency checks of the available software, such as `scram` [114], a software configuration and management tool. 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. Take the *TauRoast* application as an 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.* It is surprising that 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, as shown in Table 3.1, the CMS data stored on the HDFS filesystem has the total size of 11.6TB, 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 named but never actually accessed. There are two possible reasons for these unused dependencies. First, end users are accustomed to missing dependencies and thus get in the habit of adding commonly used software, whether it is needed or not. Second, some dependencies may be added and only used at the early stage of an analysis. As the analysis continues, these dependencies become not necessary any more. However, the user may forget to remove them from the analysis script or does not think it is a problem to keep them in the analysis script.

- *Rapid Changes in Dependencies.* Over the process of several months from collecting the initial email into a script until the *TauRoast* application got preserved, the computing environment where the application ran 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.

3.4 Refinements to Make Scientific Applications Reproducible

It is clear that the email, 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 (Table 3.1), 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. Therefore, a more structured approach is needed.

Figure 3.2 shows the three-stage refinements to convert the original email into an executable and preservable object. In each step of evolution, the dependencies of the artifact are made more explicit and available for automated processing.

The first-stage refinement: as noted in the previous section, the original author provided the prose instructions on how to repeat the application by email. To make it easier to reproduce the application, the first-stage refinement translates the instructions from the email into an executable script. More generally, this refinement happens after the author finishes configuring the execution environment and running the analysis, and aims to collect all the commands needed for reproducing an application into an executable format.

The second-stage refinement: the analysis script has embedded in it a number of external identifiers such as URLs pointing to external repositories and paths to networked filesystems. As a general programming practice, embedding such constants in the middle of a program is unwise. To tackle this, the second-stage refinement extracts 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. Using dependency map makes it possible to compose *abstract script* which simply refers to abstract file locations such as Git and CVMFS. Ideally, the author should not refer to any external resource by its path inside an analysis script unless it is indicated in a dependency map.

The third-stage refinement: in case when new data sources are used to deliver certain dependencies, the *abstract script* keeps the same, and the user only needs to update the *dependency map* accordingly. For example, in Figure 3.2, after the second-

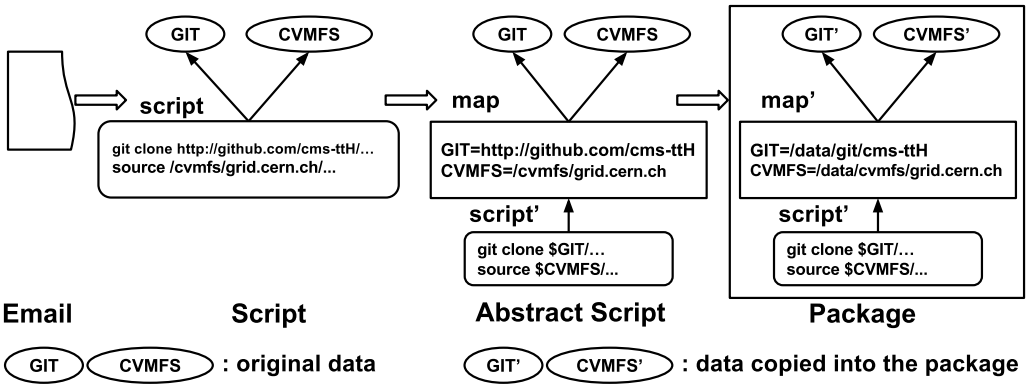

Figure 3.2. Refinements to Make Scientific Applications Reproducible

stage refinement, GIT refers to a GitHub repository, which is a network dependency; after the third-stage refinement, GIT refers to a local Git repository.

Extracting the dependencies into a dependency map introduces 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 script, dependency map, and actual dependencies into a self-contained *package*, we achieve an artifact that can be moved from place to place, as shown in Figure 3.2.

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 the dependencies mentioned in a top-level script. But, oftentimes the dependencies are embedded invisibly deep within the software stack, or are connected to the machine by the system administrator. No single party is likely to have com-

plete information about all of the dependencies. Second, the basic approach assumes that the entire dependency is actually consumed by the artifact. As 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.

3.5 Creating Minimal Execution Environment Packages: Parrot Packaging Tool

A prototype called *Parrot Packaging Tool* was implemented to assist in the measurement and preservation of implicit dependencies for complex scientific applications, which utilizes Parrot [107] to explicitly record all of the files accessed by an application. The file access information allows us to observe how much of each external dependencies is used, and what local resources are implicitly used. Using this information, a *reduced package* which contains only the files actually used by the application can be created and shared with others who are interested in reproducing 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 [44], FTP [90] and CVMFS [19]. It works by trapping an application's system calls through the Linux `ptrace` debugging interface [8], 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.

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

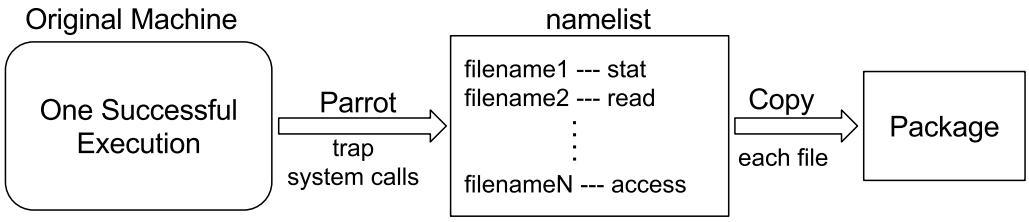

Figure 3.3. Workflow of Creating Minimal Execution Environment Packages with Parrot Packaging Tool

full URL of the object requested. However, if the connection is encrypted, this information is not available.

Figure 3.3 illustrates the workflow of creating a minimal execution environment package for an application using *Parrot Packaging Tool*. The starting point of this workflow is one successful execution of an application on the original machine. First, the original researcher executes the application under *Parrot* to generate a *namelist*, which includes the path of every accessed file and the system call used for the access. From the namelist, a package containing, and only containing, all the necessary data and software dependencies of the application can be constructed. 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.

For one execution of *TauRoast* conducted in Spring 2014, the generated namelist includes 132,047 accessed filenames, along with the system calls used to access the files, such as `open`, `stat` and `read`. With duplicate filenames removed, the namelist is reduced to 67,168 entries. Many of those entries do not exist, because they reflect attempts to search for programs and libraries in multiple places. Only 22,068 entries reflect existing files or directories.

It is worth noting that the configurations of environment variables like `PATH` and `LD_LIBRARY_PATH` have a strong impact on the namelist generated in terms of both the file access order and the accessed file number. Running the same application at different times or on a different machine may generate a different namelist. Therefore, the environment variable settings used for an application must also be preserved somehow to make the preserved artifacts reproducible.

Parrot Packaging Tool iterates over each item of the namelist, determines the process mode and replication degree according to the file location and 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. Files under `/proc`, `/dev`, `/sys` and `/net` are specific to an execution node and always ignored to be copied. Instead, during the process of rerunning an application with a preserved package, these directories will be mounted into the sandbox from the execution node.

According to the copying depth of the entries in a namelist, there are several approaches to constructing the package. In a *shallow copy*, the individual files in the namelist are copied. A directory item in the namelist causes the directory to be created and populated with empty files as placeholders to facilitate a directory listing. In a *medium copy*, the individual files are copied as before. A directory item in the namelist causes the directory to be created and populated the contents of the files in that directory, one level deep. A *deep copy* would duplicate all directories recursively, but this may have resulted in TB-sized packages, which are difficult to store and distribute currently. Therefore we will not consider this approach further. Generally speaking, the deeper the copy, the larger the package, but the more likely it can be re-purposed. Table 3.2 shows the summaries of the *shallow copy* package and the *medium copy* package for the *TauRoast* application. The *shallow* copy package includes 14273 empty files, most of which are placeholders. Using the *medium copy* approach converts 14010 of these empty files to be copied with their contents.

TABLE 3.2

SUMMARIES OF PACKAGES WITH DIFFERENT COPYING DEGREES FOR TAUROAST

File Type	**Shallow Copy**	**Medium Copy**
Number of Whole Files	1632	15642
Number of Empty Files	14273	263
Number of Directories	1549	1549
Number of Symbolic Links	4614	4614
Total Package Size	21GB	28GB

Listing 3.1 illustrates how to use *Parrot Packaging Tool* to track the files accessed during the execution of an application (Line 1), create a reduced package for the application based on the file access information (Line 2), and reproduce the application with the help of the reduced package (Line 3). In this example, the paths of all the accessed files are recorded in a namelist called `namelist1` and the execution environment variables used are collected into a file called `envlist1`. Each item in the namelist is in the format of `<file_path>|<copy_degree>`, and the `copy_degree` can be `metadatacopy` (i.e., created as a placeholder) or `fullcopy` (i.e., real content copy). During the process of creating the package using `parrot_package_create`, `envlist1` is also put into the package. After finishing the `parrot_package_create` command, a reduced package including all the really accessed files will be created at the path `/tmp/tauroast_package`, and a short summary of the execution will be returned to the user, as shown in Listing 3.2. When the package is used to rerun the application, `parrot_package_run` first loads and sets the environment variables according to the information in `envlist1`.

```
parrot_run --name-list namelist1 --env-list envlist1 ./tauroast_analysis.sh
parrot_package_create --name-list namelist1 --env-list envlist1 --package-path /tmp/
    tauroast_package
parrot_package_run --package-path /tmp/tauroast_package ./tauroast_analysis.sh
```

Listing 3.1: Usage Example of Parrot Packaging Tool

```
The packaging process has began ...
The start time is: Wed Mar  1 18:41:53 2014
Package Path: /tmp/tauroast_package
Package Size: 21G      /tmp/tauroast_package
The packaging process has finished.
The end time is: Wed Mar  1 19:11:53 2014
```

Listing 3.2: Execution Summary of Creating a Package with parrot_package_create

3.6 Package Preservation, Distribution and Deployment

Parrot Packaging Tool generates minimal execution environment packages which consist of plain filesystem trees representing the namespace and data of the preserved applications, and can be easily transformed into whatever archive file format (e.g. ZIP or TGZ) is most suitable. This makes it possible to track the integrity of the preserved artifacts with the help of hash functions such as MD5 and SHA-1, and is desirable for long-term preservation that may outlive various deployment technologies.

Once generated, application packages must be collected and curated, which involves the collaboration between researchers and archivists. To assist the reference of the preserved artifacts, it is recommended to allocate a DOI (Digital Object Identifier) for every archived package. To ease the sharing of a scientific application, the background, the problem, the goal, the solution and expected output should be documented and attached to the preserved package. How to archive private data and set access permission on archived data should also be considered.

The archived packages can be made available through public or private repositories such as Docker Hub, as shown in Figure 3.4. 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.

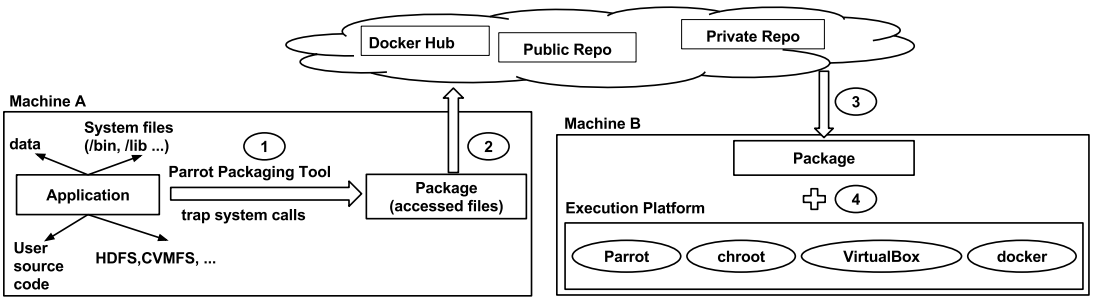

Figure 3.4. Workflow of Creating, Distributing and Deploying Packages

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 *Parrot Packaging Tool* can today be re-executed through any of the following mechanisms:

(1) Re-running the application using the `parrot_package_run` executable included in *Parrot Packaging Tool* (as shown in the 3rd line of Listing 3.1), which can dynamically construct the desired namespace and limit file accesses within the preserved package. This mechanism does not require root authority and can be used by any user.

(2) Re-running the application through *chroot*, which also works by isolating the mount namespace and limiting file accesses within the package. However, this mechanism requires the user to be root.

(3) Generating a virtual machine image from the package, which can be loaded into a local virtual machine monitor such as VirtualBox or VMware, or transferred to a cloud service provider like the Amazon EC2 and Google Computer Engine.

(4) Converting the package into a Docker image format with the `docker import` command, then reproducing the original application within a lightweight Linux container with the `docker run` command.

TABLE 3.3

TIME OVERHEADS OF THE ORIGINAL EXECUTION AND PARROT PACKAGING TOOL

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

3.7 Evaluation

This section evaluates the cost of generating a reduced package for the *TauRoast* application, compares the time overheads of executing the original script and re-executing the application with the help of the reduced package, and illustrates the correctness of the package by running the package in three different environments and comparing the result to that of running the original script.

The original script was executed on a 64-core machine running RHEL (Red Hat Enterprise Linux) 5.10 with 125 GB RAM and all the necessary remote filesystems mounted, including AFS, CVMFS, HDFS and PanFS. There are three main steps in the original script: acquiring software dependencies from different network resources, building the environment and running the analysis code. The time overhead of each step during the execution of the original script was measured and shown in the second column of Table 3.3. Executing the whole original script took about 35 minutes.

The *Software Acquisition* step acquires software from multiple locations including GitHub repositories, CVMFS and personal websites. For the purpose of reproducing

the application, there are two options on how to deliver these network dependencies. The first option is to keep the software acquisition code in the script and download all these dependencies every time the application is re-executed. The second option is to localize these dependencies first, remove the software acquisition code from the script, and then preserve them into the package like other local dependencies with the help of *Parrot Packaging Tool*. Compared with the second option, the first one is simpler. However, the execution of the application will fail if any of these network dependencies becomes inaccessible temporarily or permanently. For this reason, the second option was used in our evaluation and recommended.

With the second option, *Parrot Packaging Tool* can be used to capture the namelist (i.e., accessed file list) associated with building the environment and running the analysis code, and create a reduced package based on the namelist. Then the reduced package can be used to reproduce the application. The time overheads of obtaining the namelist, generating the package and re-executing the application are shown the third column of Table 3.3. The time overheads of obtaining the namelist and generating the package are a one-time cost. It is worth noting that the *Environment Building* step finished very fast this time. The reason for this is that the reduced package is created based on a successful execution of the application, which has already built the environment. The analysis code also got executed faster, about 7 minutes shorter than the original execution. This is because the original execution accesses data from remote filesystems like HDFS and CVMFS, however, creating the reduced package copies the dependencies from all these remote filesystems into a package, which may locate on a higher-speed disk. For example, the reduced package referred in Table 3.3 was located on an ext4 filesystem.

To verify its correctness and portability, the application was re-executed with the reduced package on the original machine, then on an independent KVM virtual machine facility provided by the Center for Research Computing at the University

TABLE 3.4

PERFORMANCE OF EXECUTING THE REDUCED PACKAGE ACROSS DIFFERENT MACHINES

Machine Type	**OS Version**	**Cores**	**Mem**	**Execution Time**
Original Machine	RHEL 5.10	64	125GB	13min 04s
KVM (Notre Dame)	CentOS 5.10	4	2GB	21min 38s
Xen (EC2)	RHEL 5.9	16	60.5GB	13min 30s

of Notre Dame, and again on a Xen-based virtual machine from the Amazon EC2 platform. Table 3.4 shows the configurations of the three machines used for re-running the package and the execution time of each run. While performance varied, the same output was obtained each time, without reference to anything outside the reduced package.

3.8 Conclusion

This chapter presents the challenges involved in reproducing complex scientific applications through a case study of reproducing one complex high energy physics application called *TauRoast*, explains a series of refinements to make complex scientific applications preservable, and presents a prototype called *Parrot Packaging Tool* to assist in preserving and reproducing complex applications. Through tracing the system calls made by an application, *Parrot Packaging Tool* can track all the accessed files during the execution of the application, generate a reduced package which includes all the accessed files. Then the package can be used to reproduce the application. To evaluate the prototype, *Parrot Packaging Tool* was utilized to create a reduced package for the *TauRoast* application, which reduced the 166.8TB data and

software dependencies of the application into a 21GB package. The reduced package was then tested in three different physical and cloud environments to reproduce the application, and functioned correctly in all these three tests.

Like *Parrot Packaging Tool*, there are other tools such as CDE [51], PTU [89] and ReproZip [32], which work via system call trapping and only preserving the really used files. However, these tools all preserve an application as it is and do not give any guidance or recommendations on how to design scientific applications in a reproducible way. This work separates data dependencies from analysis scripts by introducing *dependency maps* and *abstract scripts*. By extracting data 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.

Parrot Packaging Tool and similar tools allow the new user to exactly repeat the original experiment, but have the following drawbacks:

First, the preserved packages are often very complicated and it is difficult for a new user to understand the structure and behavior of the packages well enough to extend the original work. Exactly repeating an experiment is useful because it allows the original author and other researchers to verify the experiment results, however, is not the total aim of science and research. Researchers seldom are just interested in repeating an experiment exactly, instead they are often more interested in understanding the configuration details and the components of an experiment, tuning the experiment settings by testing it on new data sets or new analysis codes.

Second, dependencies shared by multiple applications, such as operating system images, shared libraries and shared data sets, are preserved in multiple packages repeatedly. The archival hosting the preserved packages will end up with lots of

duplicate data, which wastes the storage space. This also decreases the storage space utilization on an execution node where multiple scientific applications need to be reproduced with the preserved packages.

Third, these tools would fail on scientific applications with huge data dependencies. Take the *TauRoast* experiment as an example, if the author specifies 1TB data to analyze in the analysis script, the reduced package will be getting crazily huge.

These drawbacks exist because all these tools allow the users to configure their execution environments, run their applications, and then preserve the mess after the users finish their work. To make it easier to reuse the preserved artifacts and extend the original work, a more organized approach is needed.

CHAPTER 4

REPRODUCING SINGLE-MACHINE SCIENTIFIC APPLICATIONS WITH EXECUTION ENVIRONMENT SPECIFICATIONS

4.1 Introduction

The execution environments utilized in computational science are complex, including hardware, kernel, OS (Operating System), software, data, environment variable settings and network topology. There are frequent updates and modifications in both software and hardware. Researchers may not even be aware of all the details of the software stack used for their experiments [97] and documenting the complex web of dependencies that are common in modern software environments would be a daunting task. Unless the full execution environment can be preserved in a timely manner, even the original authors will eventually have no way to reproduce their experiments.

Virtual machines [65] and virtual appliances [57] were used to wrap up the whole software stack of an experiment into a VMI (Virtual Machine Image), which can then be used to reproduce the experiment. However, this method may fail when the software stack is too large [76]. A minimal execution environment package including only the files accessed by an experiment reduces the size of preserved artifacts to some degree, but does not solve the problem fundamentally. Both methods are not space-efficient, because common dependencies (e.g., shared libraries) are archived into different VMIs or packages repeatedly. In addition, the context and structure of preserved artifacts created by both methods are complicated and not well-documented, which makes it difficult to understand the configuration of the original experiment and extend the original work by testing new software or data sets.

Research Objects [15] was proposed to aggregate the data, methods and people involved in an experiment to facilitate the reproducibility of scientific results. However, Research Objects only bundle together the necessary resources and are not directly executable. Dynamic documents [46, 82] and reproducible papers [45] were designed to integrate the text, code, data and other auxiliary materials to make it easier to reproduce the computations. However, they mainly focus on the software and data dependencies, and does not include the hardware, kernel and OS dependencies.

This chapter presents a framework to help researchers track, create and preserve the execution environments for their experiments. The framework includes three parts - the Umbrella specification, the Umbrella execution engine and the Umbrella archiver. The Umbrella specification allows the user to specify all the details of a comprehensive execution environment - including hardware, kernel, OS, software, data, environment variables, command and output - through a lightweight, persistent and deployable JSON-format file. The Umbrella execution engine creates the execution environment specified in an Umbrella specification file using sandbox techniques like virtual machines (e.g., VMware [111]), Linux containers (e.g., Docker [80]) and user-space tracers (e.g., Parrot [107]). The Umbrella execution engine can utilize computing resources from both the local machine and cloud computing services like Amazon EC2 [4]. The Umbrella execution engine also allows common dependencies like base OS images to be shared by different sandboxes concurrently on an execution node. The Umbrella archiver allows the users to archive their execution environments into persistent storage services like Amazon S3 [85] and OSF (Open Science Framework) [7].

With this framework, the original author can make an experiment reproducible by archiving its OS, software and data dependencies using the Umbrella archiver (written in Python 2.6), and sharing the Umbrella specification which specifies all the dependencies and how these dependencies should be combined together into a unified

sandbox during runtime. Any other researcher having the Umbrella specification, the proper access permission to the involved dependencies and the Umbrella execution engine (written in Python 2.6) can easily reproduce the experiment.

Three applications from epidemiology, scene rendering and high energy physics are used to evaluate the framework. For each application, the following aspects are evaluated: the size of the Umbrella specification, the time and space overheads of creating execution environments using Umbrella, and the storage effectiveness of Umbrella for allowing dependencies to be shared by multiple execution environments concurrently. To further improve the reproducibility of scientific applications, *CurateND* [2], an archival system provided by the Hesburgh Libraries at the University of Notre Dame, was used to archive the dependencies of each evaluated application, create a DOI (Digital Object Identifier) [86] for the application, and provide an overview page including references to the Umbrella specification, experiment output and other auxiliary materials.

4.2 Why Is It So Difficult to Reproduce Experiment Results?

To understand how the complexity of execution environments makes it difficult for researchers to share their experiments, let us examine the typical workflow of scientific research.

The computing resources used by most research institutions are maintained by professional system administrators, who are responsible to install and upgrade OS and system-level software periodically and manage user accounts. As a researcher, when Alice joined her new research group, she was given access to some computing resource, let us say, a server with the hostname of `lab01.phy.research.org`. As a non-root user, Alice installed several pieces of scientific software provided by the software community in her research domain into her home directory, configured some environment variables for her convenience, wrote her own analysis script, named

`analysis.py`, using `/usr/bin/python`, which happened to be Python 2.7. Then she downloaded the datasets from the Internet and kept them locally under the directory `/home/alice/data`, ran the experiment and got the experiment results, which were converted into beautiful figures. Finally, these figures were put into her academic paper and submitted to an academic conference.

Once the paper was accepted, Alice was happy and moved on to the next challenge in her research. Everything looked great until three months later another researcher, Bob, emailed her and wanted more instructions on how to reproduce the experiment published in her paper. Telling Bob that she ran the experiment on the server `lab01.phy.research.org` would not help anything, because it would be unrealistic to give access to the server to every researcher who wants to reproduce her experiment.

Alice searched her file system for the Python script, `analysis.py`, and was relieved to find it. She shared `analysis.py` with Bob and expected him to tell her that the experiment can be reproduced successfully. However, the news from Bob was both disappointing and surprising. The problems Bob encountered during his attempt of running `analysis.py` are:

- `analysis.py` depends on the setting of the environment variable `SIMCOUNT`;
- `analysis.py` expects an input file located at `/home/alice/data/file1`;
- `analysis.py` attempts to utilize an executable named `sim_sort`;
- the output of running `analysis.py` overflows Bob's memory and disk;
- `/usr/bin/python` on Bob's machine is Python 3.0, which is not backwards compatible with Python 2.7.

Unfortunately, Alice forgot to preserve the `SIMCOUNT` setting used for her paper and deleted the directory `/home/alice/data` by accident. The source code of `sim_sort` is under version control via Git and can be found, however, Alice forgot the

commit id used for her paper. As for the memory and disk overflow, Alice realized she should have told Bob the experiment requires 6GB memory and 20GB disk space.

Although bad enough, these are only the problems relevant to the dependencies directly configured by Alice. In the background, the system administrators need to update, sometimes even upgrade, the kernel, OS and system-level software periodically. Every several years, the hardware equipment may become obsolete enough to be replaced. What's more, Alice's experiment may count on some network resources from third-party websites. Any change about these local and remote dependencies may result in the failure of reproducing Alice's experiment, no matter by herself or others.

4.3 Tracking, Creating and Preserving Execution Environments: Umbrella

Given the importance of preserving the comprehensive execution environment of an experiment for its reproducibility and the complexity of figuring out all the details about the environment, this section proposes a framework to help researchers improve the reproducibility of their research. The framework achieves this through three mechanisms:

- allows the user to specify all the necessary details about a comprehensive execution environment - from hardware, kernel and OS all the way up to software, data and environment variables (section 4.3.1);
- creates the specified execution environment using sandbox techniques like VMs, Linux containers and user-space tracers (section 4.3.2);
- helps the user archive OS, software and data dependencies in the first place (section 4.3.3).

4.3.1 Tracking Execution Environment: Umbrella Specification

Umbrella allows a user to specify a comprehensive execution environment through a JSON-format file independently of any deployment technology. The whole execu-

tion environment is specified through multiple sections, each section corresponds to a special aspect of the environment and may further include several subsections. Within each (sub)section, various attributes can be specified through `key:value` pairs. Listing 4.1 illustrates the Umbrella specification for a Povray ray tracing application. The `hardware` section specifies the requirements about the CPU architecture, the core number, the memory and disk consumption. The `kernel` section specifies the required kernel name and version. The `os` section specifies the name and version of the required OS image, which includes the basic root filesystem except for the kernel. The `software` section and the `data` section specify respectively the required software and data dependencies, which may include multiple subsections, each corresponding to a single dependency. The `environ` section allows the user to specify the environment variable settings. The `cmd` section specifies the command line used to run the experiment. The `output` section specifies the location of the experiment results.

The `os` section and each subsection under the `software` and `data` sections provide detailed information of each dependency, including the resource location (the `source` attribute), the fixity of the resource (the `checksum`, `size` and `uncompressed_size` attributes), how to use the resource (the `format` and `action` attributes) and the mountpoint at runtime (the `mountpoint` attribute). The `format` attribute specifies whether the resource is a plain file (`plain`) or a gzipped tar file (`tgz`). The `action` attribute specifies how the resource should be used during runtime - used directly (`none`) or uncompressed first (`unpack`).

The `source` attribute provides a list of resource URLs for each dependency. The resources may come from local filesystem, public web servers, Amazon S3, OSF, Git Repositories, or CVMFS (CernVM File System) [19] (as shown in Table 4.1). Resources from local filesystem can be utilized directly. Resources from public web servers are downloaded via the HTTP or HTTPS protocol. Resources from Amazon S3, OSF and Git Repositories, may have public or private access permissions, and

```
{
  "description": "A ray-tracing application which creates video frames.",
  "hardware": {
    "arch": "x86_64",
    "cores": "1",
    "memory": "1GB",
    "disk": "3GB"
  },
  "kernel": {
    "name": "linux",
    "version": ">=2.6.18"
  },
  "os": {
    "name": "redhat",
    "version": "6.5",
    "mountpoint": "/",
    "source": ["http://ccl.cse.nd.edu/.../redhat-6.5-x86_64.tar.gz"],
    "format": "tgz",
    "action": "unpack",
    "checksum": "669ab5ef94af84d273f8f92a86b7907a",
    "size": "633848940",
    "uncompressed_size": "1743656960",
    "ec2": {
      "ami": "ami-2cf8901c",
      "region": "us-west-2",
      "user": "ec2-user"
    }
  },
  "software": {
    "povray-3.6.1-redhat6-x86_64": {
      "mountpoint": "/software/povray-3.6.1-redhat6-x86_64",
      "source": ["http://ccl.cse.nd.edu/.../povray-3.6.1-redhat6-x86_64.tar.gz"],
      "format": "tgz",
      "action": "unpack",
      "checksum": "b02ba86dd3081a703b4b01dc463e0499",
      "size": "1471452",
      "uncompressed_size": "3010560"
    }
  },
  "data": {
    "4_cubes.pov": {
      "mountpoint": "/tmp/4_cubes.pov",
      "source": ["http://ccl.cse.nd.edu/.../4_cubes.pov"],
      "format": "plain",
      "action": "none",
      "checksum": "c65266cd2b672854b821ed93028a877a",
      "size": "1757"
    },
    ...
  },
  "environ": {
    "PWD": "/tmp"
  },
  "cmd": "povray +I/tmp/4_cubes.pov +O/tmp/frame000.png +K.0 -H50 -W50",
  "output": {
    "files": ["/tmp/frame000.png"],
    "dirs": ["/tmp/output"]
  }
}
```

Listing 4.1: An Umbrella Specification Example - povray.umbrella

TABLE 4.1

RESOURCE URLS SUPPORTED BY UMBRELLA SPECIFICATIONS

Resource	**Example URL**
Local Filesystem	/home/hmeng/data/input
HTTP	http://www.data.com/index.html
HTTPS	https://lab.cse.nd.edu/index.html
Amazon S3	s3+https://s3.aws.com/.../cubes.pov
Open Science Framework	osf+https://files.osf.io/v1/...7559c3a
Git Repository	git+https://github.com/.../cctools.git
CernVM File System	cvmfs://cvmfs/cms.cern.ch

are identified through their special prefixes - `s3+`, `osf+` and `git+`. When private resources from these three sources are specified in an Umbrella specification, the user needs to provide correct authentication information. Resources from CVMFS can be accessed via either the FUSE module or a user-space mount toolkit, Parrot [107].

Not all the sections are required in an Umbrella specification. If an experiment does not require any input data file, the `data` section can be ignored. The authors of an Umbrella specification may add new sections, new subsections or new attributes according to their own needs. For example, an `ec2` attribute is added into the `os` section of Listing 4.1 to specify an VMI from the Amazon EC2. It is worth noting that an AMI (Amazon Machine Image) is not unique globally, but only unique within a region [4]. Therefore, both the AMI and region information are needed to uniquely specify a virtual machine image from the Amazon EC2.

As for the software preservation format, Umbrella expects each software dependency is of binary format, not of source-code format. This decision is based on

the following three observations: first, sometimes it is difficult to obtain the source code, especially of commercial software; second, building software from source code is time-consuming; third, a successful building procedure requires special compilation toolchain and building configuration. In the case where the software building procedure needs to be reproduced, an Umbrella specification can be composed to track the compilation environment.

To make it easy to compose an Umbrella specification, we implemented a web portal (`http://umbrella.basicuserinterface.com/`) which allows the user to specify an execution environment by filling a form online and automatically computes the metadata information of a dependency once its `source` attribute is provided. The portal also provides a specification validator to help diagnose the syntax errors within a specification. Currently, the Umbrella execution engine is responsible to check the semantic errors within an Umbrella specification before creating the specified execution environment. In addition, an Umbrella specifications is often the collaborative outcome between researchers and system administrators. System administrators focus on the configurations of hardware, kernel, base OS image and system-level software. Researchers focus on software, data and all the other experiment-specific configurations.

4.3.2 Creating Execution Environment: Umbrella Execution Engine

Figure 4.1 illustrates the workflow of the Umbrella execution engine. Once an Umbrella specification is ready, either composed from scratch or downloaded from the Internet, the user can start an Umbrella job (step 1) and wait for the experiment results to be returned by Umbrella. Umbrella is responsible for parsing the specification (step 2), downloading all the missing dependencies from the locations specified in the `source` attributes (step 3), creating the execution environment by mounting all the dependencies into a unified sandbox to run the experiment (step 4)

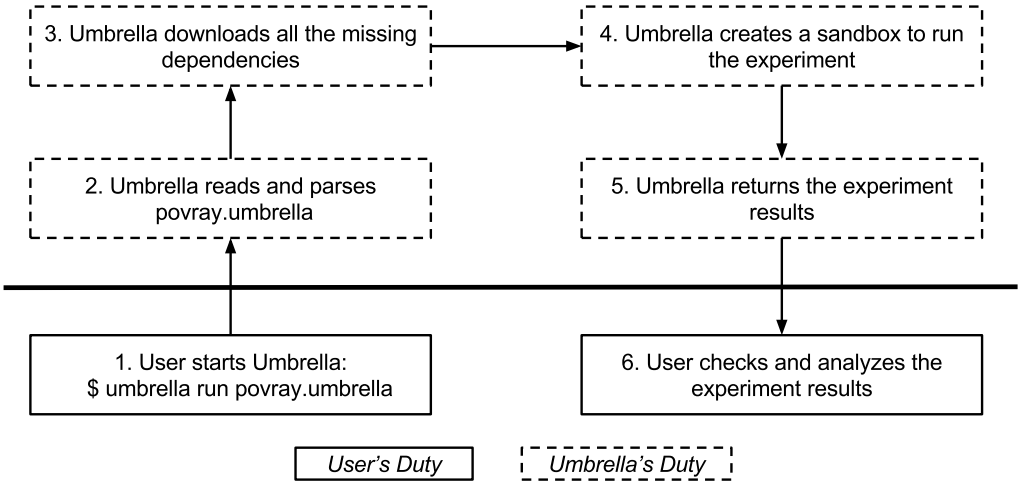

Figure 4.1. Workflow of Umbrella Execution Engine

and returning the experiment results to the location specified by the user (step 5). After this, the user can check and analyze the experiment results (step 6).

According to the matching degree between the requirements specified in the `hardware`, `kernel` and `os` sections of an Umbrella specification and the hardware, kernel and OS configurations of an execution node, different sandbox techniques can be used to create the specified execution environment. The higher the matching degree is, the lighter-weight the employed sandbox technique can be. Table 4.2 shows the available sandbox techniques for each matching degree.

4.3.2.1 Sandbox Technique - Utilize the Current OS Directly

When an execution node meets the hardware, kernel and OS requirements, it can be utilized directly to create the execution environment and every dependency will be put directly into the path specified by its `mountpoint` attribute. This method is fast because there is no virtualization being involved. However, the local filesystem may be polluted in two ways. In the case where the `mountpoint` of a dependency has not existed yet, Umbrella should create the mountpoint before putting the dependency

TABLE 4.2

SANDBOX TECHNIQUES FOR CREATING EXECUTION ENVIRONMENTS

Hardware	**Kernel**	**OS**	**Sandbox Techniques**
Yes	Yes	Yes	Utilize the current OS directly
Yes	Yes	No	OS-level Virtualization - Docker, Parrot
Yes/No	No	No	Hardware Virtualization - VMware, EC2

there. If the `mountpoint` already exists, Umbrella should first check whether the existing version is correct. When the existing version is not the required one, the user needs to be consulted about which version to keep. Due to the lack of isolation mechanisms, this method does not block any damage which may be introduced by the experiment. Therefore, it is only feasible when the behavior of the experiment is safe or the execution node is easy to recover, such as a virtual machine.

4.3.2.2 Sandbox Technique - OS-Level Virtualization

If the hardware and kernel configurations of an execution node satisfy the requirements but the OS does not, OS-level virtualization techniques can be utilized to create the execution environment. OS-level virtualization allows multiple OS instances to run simultaneously on top of a single OS kernel. Compared with hardware virtualization, this has lower execution overhead because no hardware-level instruction translation is needed. An implementation of OS-level virtualization may only deploy file system isolation (such as `chroot` and Parrot) or isolate file system, process, user, hostname and network, and set limits about memory and CPU usage (such as Docker). By isolating the root filesystem of each OS instance, OS-level virtualization avoids the risk of ruining the file system of the execution node.

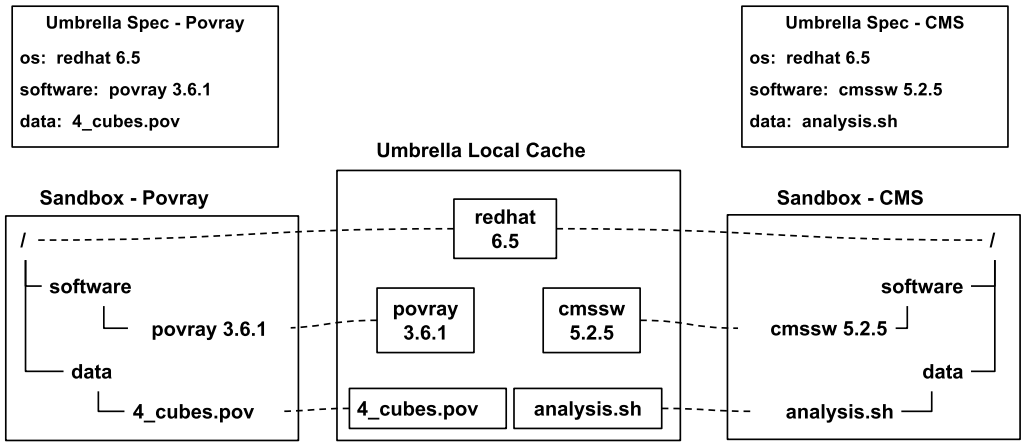

Figure 4.2. Umbrella Local Cache - Allowing Dependency Sharing

This isolation also makes it possible to share the common dependencies between sandboxes for different applications, as illustrated in Figure 4.2. The two applications - a Povray ray tracing application and a CMS application - share the same OS image (`redhat 6.5`), but each has its own software and data dependencies. Umbrella downloads all the distinct dependencies into its own local cache and mounts the dependencies into each sandbox during runtime. In this scenario, the Umbrella local cache only keeps a single copy of the OS image, which will be shared by the two sandboxes. Within each sandbox, the data mounted from the Umbrella local cache can not be modified. All the modifications during runtime should be written to the mountpoint mounted from a location outside of the Umbrella local cache.

4.3.2.3 Sandbox Technique - Hardware Virtualization

When the hardware or kernel configurations of the execution node do not satisfy the requirements, hardware virtualization can be used to emulate the target computing environment in a VM, and create the execution environment within the VM. Compared with OS-level virtualization, Hardware virtualization involves more execution overhead because each instruction within the VM needs to be translated into

an instruction on the host via a virtual machine monitor (i.e., hypervisor). However, since the virtual machine already satisfies the hardware, kernel and OS requirements, the sandbox can be constructed directly inside it, as discussed in section 4.3.2.1.

This sandbox technique can be implemented in two ways depending on where a virtual machine is hosted. If the current execution node has a hypervisor such as VirtualBox [112] or VMware [111] installed, it can be used directly to launch the required VM and finish the Umbrella job within it. If the current execution node has not installed any hypervisor or the user prefers not to do the test locally, cloud computing platforms such as Amazon Elastic Computing Cloud (EC2) and Google Compute Engine (GCE) can be utilized. The composers of Umbrella specifications should specify the required cloud platform and virtual machine image, as illustrated by the `ec2` subsection under the `os` section in Listing 4.1. Umbrella responds to communicate with the cloud platform to start a VM, send the Umbrella job to the VM and launch the Umbrella job on the VM locally. Then the VM creates the execution environment and finishes the job. Finally, the experiment results will be sent back to the local machine and put into the user-specified location. Figure 4.3 illustrates how Umbrella can utilize Amazon EC2 to finish a job.

Our experience with the Amazon EC2 suggests that, in spite of all the benefits it provides for allowing researchers to share their work, the Amazon EC2 service alone cannot achieve reproducibility for the following reasons. First, an AMI is not globally unique, instead only unique within a region. Copying an AMI from one region to a different region would result in an AMI with a different identifier. To specify an AMI for the purpose of reproducibility, both the AMI and the region where it is located are needed. Second, the username and password settings of virtual machine images provided by the Amazon EC2 service vary according to the OS distributions, which is inconvenient for sharing images between researchers. Third, the instance types supported by the Amazon EC2 service change depending on the hardware

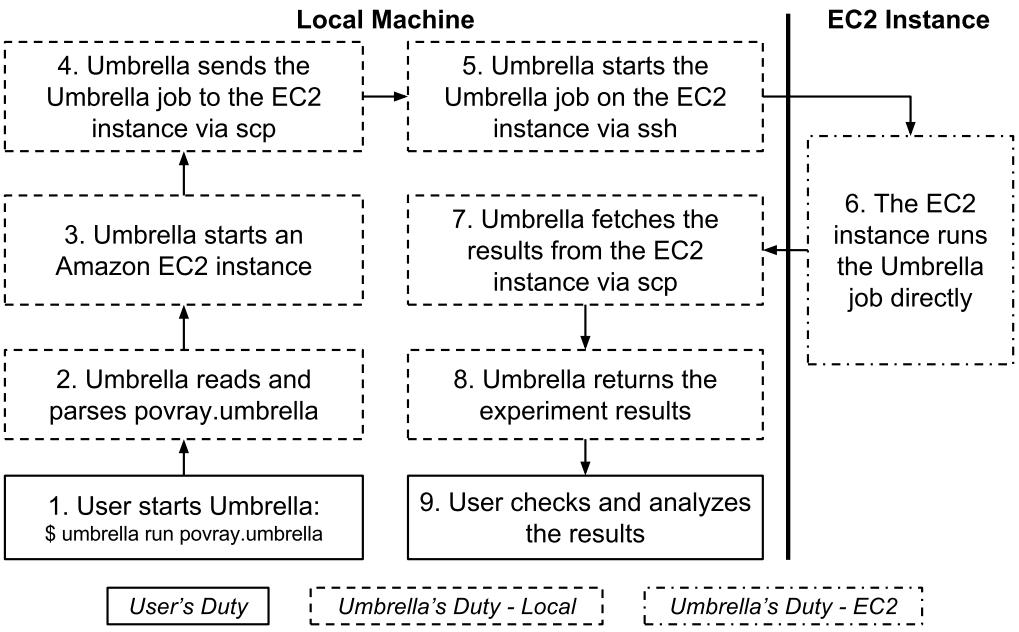

Figure 4.3. Workflow of Executing An Umbrella Job via Amazon EC2

costs. Some instance types may become obsolete as time goes by, which makes it challenging to conduct performance-oriented reproducible research.

4.3.3 Preserving Execution Environment - Umbrella Archiver

Once the final experiment configuration is figured out, it is time to archive the involved dependencies, especially those dependencies from unreliable sources, such as local disks and some third-party websites. The Umbrella archiver was designed to help researchers archive the dependencies specified in an Umbrella specification into persistent storage services. It accepts the user credentials for the target storage services from its command line options and communicates with the target storage services via their Python bindings. By default, all the OS, software and data dependencies specified in an Umbrella specification are archived into the storage system. The researchers may mark off the already-archived dependencies with a JSON field, `upload:false`. Once the archiving process is done, Umbrella will update the re-

source URLs of all the relevant dependencies to the reliable ones and generate a new Umbrella specification.

Currently, Umbrella supports two persistent storage services: Amazon S3 and OSF. Archiving an execution environment to Amazon S3 creates a new S3 bucket, uploads each unreliable dependency into the bucket and finally uploads the updated Umbrella specification into the bucket. Then researchers can publish and share the S3 link of the new Umbrella specification. Archiving an execution environment to OSF creates a new OSF project, archives each unreliable dependency as an OSF file under the OSF project and finally uploads the updated Umbrella specification into the OSF project. Then researchers can publish and share the OSF URL of the new Umbrella specification. The Umbrella archiver also allows researchers to set the access permission on uploaded OSF and S3 resources.

4.4 Two Example Workflows of Conducting Reproducible Research

This section describes two typical scenarios where the framework proposed in section 4.3 can be used to facilitate the reproducibility of scientific research and provides the detailed workflow of how to use the framework to achieve this. In both scenarios, Alice is the original author of the experiment and Bob is another researcher who wants to reproduce Alice's work.

In the first scenario, Alice conducts her experiment on her local machine and all the dependencies are from the local machine. Alice first composes an Umbrella specification, `povray_local.umbrella` in Figure 4.4, to describe the execution environment of her experiment. Then Umbrella is used to create the specified execution environment and execute the experiment. Whenever Alice wants to tune the experiment settings, she always first updates the execution environment specification. By tracking the execution environment as the research process goes and even before every real execution starts, Alice is always sure about the environment configurations

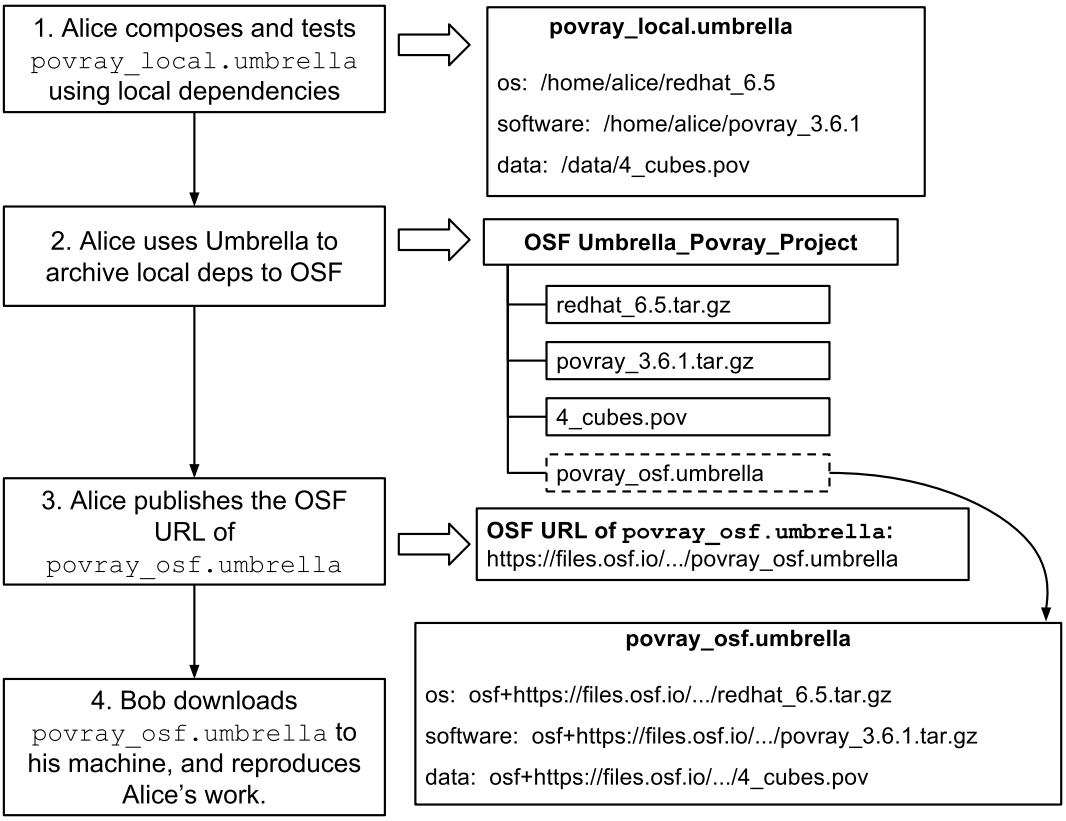

Figure 4.4. Conducting Reproducible Research Using Umbrella - Local + OSF

of a successful execution. After Alice finishes her experiment, she uses Umbrella to create a new OSF project and upload all the local dependencies into the OSF project. Umbrella also creates a new specification with all the dependencies from OSF, `povray_osf.umbrella`, which is also uploaded into the OSF project. Then Alice attaches the OSF URL of `povray_osf.umbrella` to her paper. After this, any change or accident happened to the local machine used by Alice will not jeopardize the reproducibility of the experiment. If Bob reads Alice's paper and wants to reproduce the experiment, he can download `povray_osf.umbrella` using the OSF URL in the paper and reproduce Alice's work. Figure 4.4 illustrates the whole workflow.

In the second scenario, Alice wants to conduct her experiment using Amazon EC2. To avoid downloading the dependencies of the experiment from the outside Internet

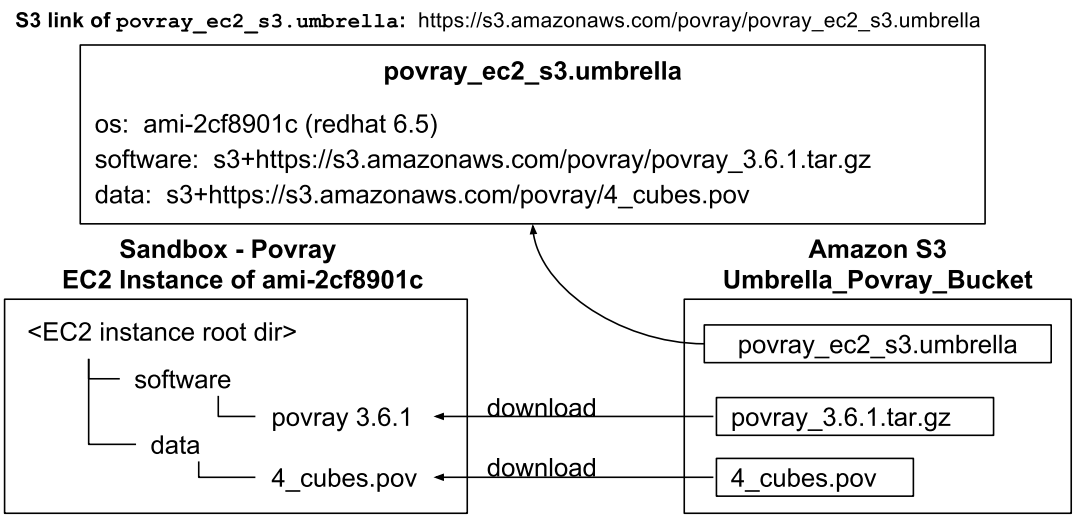

Figure 4.5. Conducting Reproducible Research Using Umbrella - EC2 + S3

to EC2 instances repeatedly, Alice stores the software and data dependencies inside Amazon S3. Alice composes an Umbrella specification, `povray_ec2_s3.umbrella` in Figure 4.5, to describe the execution environment of her experiment. Then Alice uses Umbrella to communicate with AWS (Amazon Web Services) and execute the experiment using an EC2 instance (section 4.3.2.3). During runtime, all the dependencies are downloaded from Amazon S3 into the EC2 instance and then used to create the execution environment. When Alice is ready to publish her experiment result, she just needs to put `povray_ec2_s3.umbrella` into Amazon S3 and publishes its S3 link in her paper. By doing so, Bob can obtain a copy of `povray_ec2_s3.umbrella` easily and reproduce Alice's work.

In both scenarios, Bob can first read the Umbrella specification file shared by Alice to understand the hardware and software requirements of the experiment, and then decide where to reproduce it and which sandbox mode to use to reproduce it. This helps Bob avoid the time overhead of asking Alice about the environment variable settings and input files, running out of memory and disk space by accident, and the failure caused by incompatible versions of Python.

4.5 Evaluation

A prototype of the framework was implemented using `Python2.6` as a Python script, `umbrella`, which supports the features described above - tracking, creating and preserving execution environments of scientific applications. As for creating execution environments, four sandbox modes using different sandbox techniques are currently supported: the `destructive` mode, which utilizes the current OS directly; the `parrot` mode and `docker` mode, which are two examples of OS-level virtualization; the `ec2` mode, which is an example of hardware virtualization utilizing Amazon EC2. The `ec2` and `destructive` modes are usually used together: the local Umbrella uses the `ec2` mode to start an VM and the remote Umbrella running on the VM uses the `destructive` mode to run the experiment.

To evaluate the framework, `umbrella` was utilized to reproduce three scientific applications from epidemiology, scene rendering and high energy physics (section 4.5.1). The Umbrella specification file sizes (section 4.5.2), the space and time overheads of creating execution environments using different sandbox techniques (section 4.5.3), and the effectiveness of the Umbrella local cache for allowing dependencies to be shared by multiple execution environments (section 4.5.4) are evaluated.

The evaluation results show that, with the help of Umbrella, an experiment can be reproduced successfully by the original author and others with different sandbox techniques, even though the overhead and performance may vary slightly.

4.5.1 Applications Evaluated

Three applications were used to evaluate the framework: an OpenMalaria application from epidemiology, a Povray ray tracing application and a CMS (Compact Muon Solenoid) application from high energy physics.

OpenMalaria aims to develop a model to simulate the potential effects of the introduction of pre-erythrocytic malaria vaccines [99]. It uses individual-based stochastic

simulations of malaria epidemiology to predict the impacts of interventions on infection, morbidity, mortality, health services use and costs. The OpenMalaria application used here does a VecNet baseline simulation with increased population size for more precise results. The application takes an xml-format input file which describes the place being simulated - human population distribution, entomology of vectors, effectiveness of health system in the area and optionally interventions applied to control malaria. The outputs of the application include a file with the size of 39KB capturing every timestep of a simulation (`ctsout.txt`) and a file with the size of 51KB aggregating data into configurable-size lumps (`output.txt`).

Povray is a ray tracing program which renders 3D graphics from text-based scene descriptions that describe all the details of a scene, including the camera, lights, plane and objects. All povray objects are described by mathematical functions and represented internally using their mathematical definitions. The povray application used here has two data dependencies: a scene description file (`.pov`) and a `include` file containing the mathematical functions of Rubik's Cube (`.inc`). The output of the application is a `png` file with the size of 16MB.

The CMS experiment investigates the most basic building blocks of matter by observing collisions of protons at near light-speed. Large numbers of collisions must be simulated and analyzed statistically to develop accurate models of the underlying physical processes. We applied the Umbrella framework to a step in this simulation process, in which collisions between protons are simulated. The results of each collision are generated, based on the model under test, and recorded. The input consists of a Python configuration file describing the process to be investigated and the output is a 96MB enhanced ROOT file containing the recorded particle descriptions. Due to metadata recorded at runtime which includes timestamps, all outputs will differ in a non-deterministic way. Because the psuedorandom seed is fixed, however, the generated output of a given configuration will always be identical.

TABLE 4.3

DEPENDENCIES OF EVALUATED APPLICATIONS

Application	**OS Deps**	**Software Deps**	**Data Deps**
OpenMalaria	CentOS 6.6 (69MB/218MB)	openMalaria (2.9MB/13MB)	.xml (28KB)
		.rpm packages (209MB)	.csv (<1KB)
		epel.repo (<1KB)	.xsd (196KB)
Povray	RedHat 6.5 (605MB/1.8GB)	povray (1.5MB/2.9MB)	.pov (1.8KB)
			.inc (28KB)
CMS	RedHat 6.5 (605MB/1.8GB)	cmssw (1.3GB)	.sh (<1KB)
		parrot (23MB/71MB)	

Table 4.3 illustrates the dependencies of each application and the size of every dependency. Two archive formats are used for the preservation of these dependencies: plain files (e.g., epel.repo) and gzipped tar files (e.g., CentOS 6.6). Both the compressed and uncompressed sizes of a gzipped tar file dependency are listed in Table 4.3. The size information of a plain-format dependency is in the format of `(size)`. The size information of a gzipped tar file dependency is in the format of `(compressed_size/uncompressed_size)`. All the dependencies are archived in *CurateND*, a campus archival system for curating Notre Dame's Research and scholarship for study throughout time provided by the Hesburgh Libraries at the University of Notre Dame [2].

Although only the evaluation results for three applications are shown in this section, the framework does not bind itself with any specific field and is broadly applicable to scientific applications from other fields.

TABLE 4.4

UMBRELLA SPECIFICATION FILE SIZES

Application	OpenMalaria	Povray	CMS
Umbrella Spec Size	3.3KB	2.4KB	1.9KB

4.5.2 Umbrella Specification File Sizes

The execution environment for each application is tracked via an Umbrella specification file, which specifies the hardware, kernel, OS, software dependencies, data dependencies, environment variables, analysis command and application output. Table 4.4 illustrates the Umbrella specification file sizes for these three applications. An Umbrella specification only includes resource identifiers and other metadata information, rather than the real content of any dependency. Therefore, in contrast with the dependency sizes shown in Table 4.3, an Umbrella specification file itself is only a few kilobytes.

To allow these applications to be reproduced with different sandbox techniques, the `os` section of each specification specifies an AMI (Amazon Machine Image) for the `ec2` sandbox mode and an OS image in the format of gzip tar file for the `parrot` and `docker` sandbox modes.

4.5.3 Overheads of Creating Execution Environments

Umbrella can create the execution environment specified in an Umbrella specification file with different sandbox techniques. Table 4.5 illustrates the time and space overheads of reproducing these three applications using three sandbox techniques - `parrot`, `docker` and `ec2`. Even if the time and space overheads of reproducing each application vary according to the sandbox techniques and computing resources used,

TABLE 4.5

TIME AND SPACE OVERHEADS OF CREATING EXECUTION ENVIRONMENTS

Application	OpenMalaria	Povray	CMS	Permission	Location
parrot	N/A	65min (2.40GB)	79min (2.39GB)	non-root	locally
docker	57min (1.53GB)	68min (4.11GB)	82min (4.19GB)	root	locally
ec2 - m3.medium	113min (225MB)	130min (4.4MB)	211min (94MB)	non-root	remotely
ec2 - m3.large	58min (225MB)	65min (4.4MB)	108min (94MB)	non-root	remotely

they all generate the correct output. The `parrot` and `docker` sandbox modes are tested on the same machine (hardware: x86_64, kernel: Linux 2.6.32, OS: redhat 6.7).

The time overheads of different sandbox modes vary for two main reasons. First, sandbox techniques based on hardware virtualization (`ec2`) involve higher overheads than sandbox techniques based on OS-level virtualization (`parrot` and `docker`). Furthermore, the overheads of different OS-level virtualization techniques vary according to the isolation degree and the implementation details. Second, Umbrella allows the hardware and kernel requirements to be specified as a range, not limited as a single value. For example, the user can specify the CPU core number to be at least 2 and the memory space to be at least 3GB. This design decision was made because we believe, from a long-term perspective, the aim of reproducibility is first to get the correct result, then to care about the performance.

These three applications are all single-threading and CPU-intensive, therefore the time overheads are mainly determined by the CPU performance of the execution nodes. The `parrot` and `docker` sandbox modes are tested on the same machine, the difference between the time overheads of each application under these two modes are mainly due to their implementation details. As a user-space tracer, `parrot` works by

trapping each system call and redirecting the file access if necessary. `Docker` mainly bases on several Linux kernel features including namespaces and control groups. The EC2 `m3.medium` instance type has one virtual CPU (vCPU) and the `m3.large` instance type has two vCPUs. The evaluation result suggests that the performance of each EC2 vCPU is about half of two vCPUs, which performs pretty similarly to a nonvirtualized CPU.

The space overhead of the `parrot` sandbox mode covers the total size of plain-format dependencies and gzipped tar file dependencies (both compressed version and uncompressed version). This should be identical to the summary of the dependency sizes listed in Table 4.3. The space overhead of the `docker` sandbox mode covers all the dependencies listed in Table 4.3 and an extra hard copy of the OS image dependency, which needs to be copied from the Umbrella local cache into the storage backend of Docker. The `ec2` sandbox mode only needs to download software and data dependencies into the Umbrella local cache on the EC2 instance, whose OS image already satisfies the requirement.

Comparing with the Povray ray-tracing application and the CMS application, creating the execution environment of OpenMalaria has a special step of installing several rpm packages via the package manager, `yum`, which requires the root authority. The `parrot` sandbox mode is based on a user-space mount toolkit, Parrot, therefore can not be used to create the execution environment of OpenMalaria. On the contrary, both `docker` and `ec2` sandbox modes can gain the root authority and can be used here to install rpm packages with `yum`.

The `parrot` sandbox mode can be used by both root and non-root users, because *Parrot* is a user-space tracer. Using the `docker` sandbox mode requires the user to be root or be added into the docker group to communicate with the docker daemon. The docker daemon always runs as the root user and adding a non-root user into the docker group also requires root authority. The `ec2` sandbox modes does not

necessarily require the root authority, but requires the EC2 python module, `boto3`, to communicate with the EC2 resources.

In summary, the decision of choosing the right sandbox technique to reproduce an application depends on the following three factors: whether the user has root authority on an execution node, where the user wants to reproduce an application (locally or remotely), and whether the application itself involves privileged operations.

4.5.4 Effectiveness of Umbrella Local Cache

Umbrella tries to cache a dependency into its local cache when it is first required, so that the following experiments can reuse the local copy inside the local cache without downloading it repeatedly from the Internet. This saves both the time and economic cost of reproducing an experiment. Table 4.6 shows the changes of the Umbrella local cache size as different experiments or different software/data dependencies are tested. The CMS experiment and the Povray experiment are used here, which share the same OS image dependency. All the tests here were done with the `parrot` sandbox mode on the same machine (hardware: x86_64, kernel: Linux 2.6.32, OS: redhat 6.7).

Originally, the Umbrella local cache is empty. When the CMS experiment is tested, all its dependencies, totally 2.39GB, are downloaded into the Umbrella local cache. Next, when the Povray experiment is tested, only its software and data dependencies are missing and added into the cache, totally 4.4MB. When the researcher wants to rerun any of these two experiments, all the required dependencies have existed in the local cache and no new dependencies are needed. If the user wants to test a new software or data dependency, only the new dependencies will be downloaded from the Internet and added into the cache. Rerunning these experiments does not greatly reduce the execution time, because all the dependencies are archived in *Cu-*

TABLE 4.6

EFFECTIVENESS OF UMBRELLA LOCAL CACHE

Application (Deps Size)	**Cache Size**	**Delta (Newly Added Deps)**	**Time**
CMS (2.39GB)	2.39GB	2.39GB (all deps)	79min
Povray (2.40GB)	2.40GB	4.4MB (software & data)	64min
CMS - rerun	2.40GB	0	78min
Povray - rerun	2.40GB	0	64min
Povray - new software deps	2.40GB	4.4MB (software)	64min
Povray - new data deps	2.40GB	28KB (data)	64min

rateND, which is on the same campus network with the execution node, and the downloading speed from *CurateND* to the execution node is about 30MB/s.

4.5.5 Last Step to Enhance Reproducibility

The framework described above facilitates the reproducibility of scientific research. To go further, we created a DOI for each evaluated application, as shown in Table 4.7, using the campus archival service provided by our university library, *CurateND*. Each DOI points to an overview page hosted by *CurateND*, which includes the Umbrella specification file, the links to the Umbrella installation documentation and user manual, all the dependencies and the experiment results. The original authors can decide when to publish and share their work by setting the proper access rights to the resources referred in the overview pages. Using *CurateND*, anyone having the proper access rights can download the Umbrella specification for an experiment, install the Umbrella binary, reproduce the experiment, verify the published experiment results and even extend the original experiment.

TABLE 4.7

DOIS FOR THE EVALUATED APPLICATIONS TO ENHANCE REPRODUCIBILITY

Application	**DOI URL**
OpenMalaria	http://dx.doi.org/doi:10.7274/R03F4MH3
Povray	http://dx.doi.org/doi:10.7274/R0BZ63ZT
CMS	http://dx.doi.org/doi:10.7274/R0765C7T

4.6 Conclusion

This chapter proposes a framework facilitating the conduct of reproducible research by tracking, creating and preserving the comprehensive execution environments of scientific applications with Umbrella. Using this framework, researchers can make their experiments reproducible by specifying their execution environments in a lightweight, persistent and deployable execution environment specification, which can then be easily shared, expanded or repurposed. Researchers can also utilize the framework to archive their execution environments into persistent storage services like Amazon S3 and OSF to facilitate the sharing and publication of their experiments. The framework also provides an execution engine which allows the original researchers and any other researchers to (re)create the specified execution environments using sandbox techniques like VM, Docker and Parrot.

The framework suggests the archive unit of dependencies to be each OS image, software and data. Different applications sharing the same dependencies can simply refer them, which avoids preserving the same dependencies multiple times and saves the storage space in an archive. This also facilitates the management of archived data by allowing different dependencies to be updated in different frequencies and managed separately. This finer archive granularity also makes it possible to specify

an execution environment in different sections like an `os` section, a `software` section and a `data` section, which makes it easy for new users to understand the components of an experiment and extend the experiment.

The main reason why experiment results published in a paper are difficult to be reproduced is that the experiment procedure is not fully documented due to different factors such as the time pressure to move on to the next research challenges after the paper is published, lacking of funding and leaving of some key members from the team. The commonality of these factors is that the documentation procedure is delayed to the last minute. To fix this, the framework suggests that researchers document their experiment as the research goes by specifying the execution environment first and then running the experiment. Whenever there is any tunes or changes to an execution environment, the specification should always be changed first.

The Umbrella framework also tries to differentiate execution environment specifications and the tools used to create the specified execution environments. An execution environment specification should be persistent and deployable. However, the tools used to create an execution environment may experience frequent changes due to the advance of technology. During the process of this work, virtual machines were first utilized to create execution environments, then the Amazon EC2, then user-space system call tracers and Linux Containers. Even with the same technology, there may exist multiple implementations.

CHAPTER 5

REPRODUCING SCIENTIFIC WORKFLOWS WITH UMBRELLA

5.1 Introduction

Different solutions have been proposed to reproduce single-machine scientific applications. Some popular solutions include virtual machines [57], Linux Containers (e.g., Docker [80]) and user-space ptrace-based tools (e.g., CDE [51] and Parrot Packaging Tool [75]). In spite of their difference, these solutions all emphasize the importance of preserving the complete software stacks (i.e., execution environments) of scientific applications for conducting reproducible research [79].

However, lots of scientific applications are too big to be solved on a single machine, due to their huge computing and storage requirements. To solve this, *scientific workflows* [103] were designed to disseminate complex data transformations and analysis procedures into an ordered list of smaller and possibly independent tasks, which allows computing resources from clusters, grids and clouds to be utilized to accelerate the pace of scientific progress. The tasks involved in a scientific workflow are often organized into a DAG (Directed Acyclic Graph), where nodes represent tasks and files, and edges represent data flow and dependency relationship. Figure 5.1 shows a DAG including six tasks (`T1 - T6`) and six files (`F1 - F6`), which represents the simple workflow example in Figure 5.2, written in the *Makefile* language [11]. It is worth noting that this workflow is for demonstration purposes only. A real scientific workflow is usually more complex in both task number and task dependencies. It is worth noting that not all scientific workflows can be described as DAGs. This work focuses on the scientific workflows which can be described as DAGs.

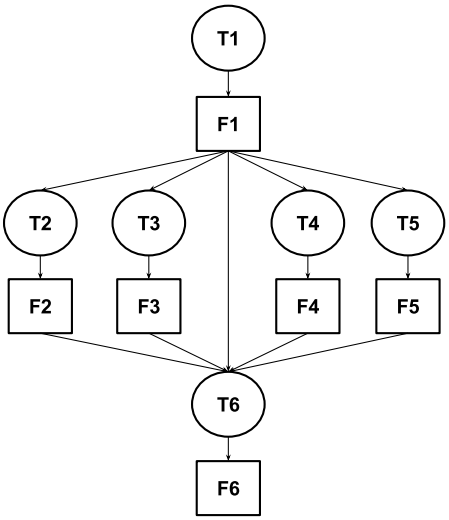

Figure 5.1. An Example Workflow in DAG

```
F6  capitol.montage.gif: capitol.jpg capitol.90.jpg capitol.180.jpg
    capitol.270.jpg capitol.360.jpg /usr/bin/convert
            convert -delay 10 -loop 0 capitol.jpg capitol.90.jpg                    T6
    capitol.180.jpg capitol.270.jpg capitol.360.jpg capitol.270.jpg
    capitol.180.jpg capitol.90.jpg capitol.montage.gif
F5  capitol.90.jpg: capitol.jpg
            convert -swirl 90 capitol.jpg capitol.90.jpg                            T5
F4  capitol.180.jpg: capitol.jpg
            convert -swirl 180 capitol.jpg capitol.180.jpg                          T4
F3  capitol.270.jpg: capitol.jpg
            convert -swirl 270 capitol.jpg capitol.270.jpg                          T3
F2  capitol.360.jpg: capitol.jpg
            convert -swirl 360 capitol.jpg capitol.360.jpg                          T2
F1  capitol.jpg: /usr/bin/curl
            curl -o capitol.jpg http://ccl.cse.nd.edu/images/capitol.jpg            T1
```

Figure 5.2. An Example Makefile: Image Rotation

Each task is specified by a rule containing two lines. The first line is in the format of **output_files: input_files***, and the second line is the command.*

To make it easy for scientists to compose and execute scientific workflows, a variety of *scientific workflow systems* have been developed [116], such as Taverna [83], Swift [115], Pegasus [37] and Makeflow [11]. The end-users of these workflow systems only need to specify an ordered list of tasks. The workflow systems respond to communicate with execution engines, schedule tasks to the underlying computing resources, manage data sets and deliver fault-tolerance.

Ideally, scientific workflows should improve the reproducibility of scientific applications by making it easier to share and reuse workflows between scientists. However, scientists often find it difficult to reuse others' workflows, which is known as *workflow decay* [53]. For example, myExperiment [48] is designed to be a sharing social website which allows scientists to share their workflows, however, a study of Taverna workflows on myExperiment shows that 80% of the workflows on the site could not be reproduced [119].

Guidelines on how to design reproducible scientific workflows were proposed in [47, 53] and *Research Objects* [15] were used to provide detailed metadata information about workflows [53]. However, these guidelines focus on the specification layer of scientific workflow systems and ignore the other components of scientific workflow systems. As shown in Figure 5.3, scientific workflow systems usually include multiple layers - the workflow specification layer, the specification parser layer, the task scheduler layer and the computing resource layer.

Depending on the scientific workflow systems used, scientists have different levels of control over the underlying execution environments. Pegasus [37] and Swift [115] allow scientists to compose *abstract workflows* without worrying about the details of the underlying execution environments, which means sysadmins must respond to the cumbersome job of configuring computing resources to meet all the requirements of different workflows. Makeflow [11] allows executables to be specified in workflow specifications and delivered to execution nodes at runtime, such as `/usr/bin/convert`

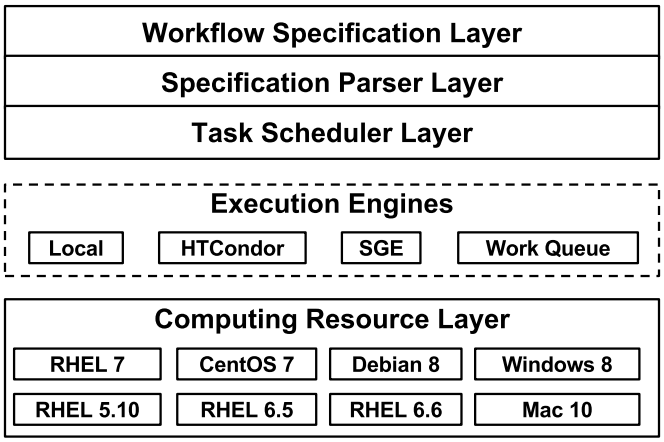

Figure 5.3. Layers of Scientific Workflow Systems

and `/usr/bin/curl` in Figure 5.2. This is simple but not always correct, because executables may be sent to execution nodes with incompatible execution environments. To fix this, Makeflow allows scientists to specify a Docker image [80] containing the required execution environment and delivers the image to execution nodes at runtime [120]. This gives scientists more control over the execution environments, but ends up with gigantic images which are expensive to store.

This chapter explores the challenges in reproducing scientific workflows and proposes a framework for facilitating the reproducibility of scientific workflows by giving scientists complete control over the execution environments for their workflows and integrating execution environment specifications into scientific workflow systems. This framework allows the dependencies of execution environments to be archived separately in finer granularity instead of gigantic all-in-one images. A prototype of the framework was implemented by integrating *Umbrella* [77], the execution environment creator explained in Chapter 4, into *Makeflow* [11], a scientific workflow system which can utilize computing resources from a HTCondor cluster [108].

Two bioinformatics scientific workflows, *BLAST* and *BWA*, are used to evaluate the framework. The execution environment for each workflow is specified as an

Umbrella specification file and sent to the execution nodes in the Notre Dame HTCondor pool, where *Umbrella* is used to create the required execution environment to run the tasks. For each workflow, the size of the Umbrella execution environment specification, the time and space overheads of creating execution environments using Umbrella, and the heterogeneity of execution nodes contributing to each workflow, are evaluated.

It is worth noting that the framework proposed in this chapter does not solve the problem of reproducing scientific workflows completely. Our framework mainly focuses on how to improve the reproducibility of scientific workflows by improving the reproducibility and portability of each task within a scientific workflow. However, our framework does not cover how to reproduce the underlying execution engines, such as the HTCondor computing framework. In addition, running scientific workflows also has its own execution environment requirements, such as the executables used to start workflows and its dependencies. However, due to the space limitation, this is not covered in this chapter.

5.2 Challenges in Reproducing Scientific Workflows

A scientific application usually takes a specific format of input and generates a specific format of output. *Scientific workflows* aim to speed up the progress of scientific applications by breaking down the complex data transformations and analysis procedures into an ordered list of smaller and possibly independent tasks, and executing the independent tasks concurrently. *Scientific workflow systems* are designed to make it easy for scientists to compose and execute scientific workflows by taking the responsibility of managing the underlying computing resources, scheduling tasks onto computing resources and hiding the complexity from scientists.

The reproducibility of scientific workflows depends on how scientific workflow systems are designed and implemented. This section explores the characteristics of

scientific workflow systems observed throughout this work, which make it challenging to reproduce scientific workflows.

- **Complexity.** Scientific workflow systems usually include multiple layers, as shown in Figure 5.3. The complexity of different layers vary greatly. The workflow specification layer simply includes workflow languages and workflow specifications, which can be easily preserved. However, the task scheduler layer often communicates with multiple execution engines to achieve maximal speedup. The computing resource layer includes multiple software stacks (possibly thousands or even more), together with the networking connecting them together, and requires much more efforts to be preserved and reproduced.

- **Dynamics.** The stability of different layers of scientific workflow systems vary a lot. The workflow specification layer and the specification parser layer are usually very stable and tightly coupled. Adding new syntax into a workflow language would require the specification parser change accordingly. Old versions of specification parsers may fail to parse workflow specifications using new syntax features. The task scheduler layer may add support for new execution engines as new computing frameworks become popular. The computing resource layer usually experiences more frequent changes, which is especially true for opportunistic computing framework like HTCondor [108].

- **Heterogeneity.** The hardware and software configurations of machines in the computing resource layer are often heterogeneous, both across different execution engines and within a single execution engine. As an example, Table 5.1 shows the statistics of machine configurations of the Notre Dame HTCondor pool at Fall 2016, which is very different from the statistics captured at Spring 2015, as shown in Table 5.2. It is worth noting that currently the Notre Dame HTCondor pool has 7 different versions of RHEL (Red Hat Enterprise Linux). Machines in cluster computing may start with homogeneous configurations, but eventually would end up with heterogeneous configurations due to hardware replacement and upgrade. As for the filesystem directory structure, not all the Linux distributions comply the filesystem hierarchy standard maintained by the Linux Foundation [6] completely.

- **Incompatible Execution Environments on Execution Nodes.** Execution nodes in the computing resource layer often do not have the correct execution environments required by tasks. One possible solution is to ask sysadmins to install all the missing dependencies on each execution node. However, this is not scalable due to the dependency complexity and diversity of different scientific workflows. Another solution is to allow scientists to specify executables and deliver them to execution nodes together with tasks, as done in Makeflow [11]. However, this is not always correct, even if we consider two very simple and

TABLE 5.1

HETEROGENEITY OF THE ND HTCONDOR POOL (FALL 2016)

Attribute	**Description**
Machine Number	1886
Hardware Architecture	x86_64, i686
Kernel Version	Linux, Darwin, Windows NT
OS	RHEL, Mac, Windows
RHEL Versions	5.11, 6.5, 6.6, 6.7, 6.8, 7.0, 7.2
CPU Number	1, 2, 4, 8, 12, 16, 24, 32, 64
Memory Size	Min: 1002MB, Max: 251GB
Disk Size	Min: 9GB, Max: 4.7TB

TABLE 5.2

HETEROGENEITY OF THE ND HTCONDOR POOL (SPRING 2015)

Attribute	**Description**
Machine Number	4157
Hardware Architecture	x86_64, i386, i686
Kernel Version	Linux, Darwin, Windows NT
OS	RHEL, Debian, CentOS, Mac, Windows
RHEL Versions	5.5, 5.9, 5.10, 5.11, 6.4, 6.5, 6.6, 7.0
CPU Number	1, 2, 4, 8, 12, 16, 24, 32, 64
Memory Size	Min: 984 MB, Max: 1TB
Disk Size	Min: 5GB, Max: 1.7TB

TABLE 5.3

SOFTWARE INCOMPATIBILITY ACROSS DIFFERENT RHEL DISTRIBUTIONS

Executable (Shared Object Deps Num)	**RHEL5.11**	**RHEL6.8**	**RHEL7.0**
curl from RHEL5.11 (18)	✓	libcurl.so.3	libcurl.so.3
curl from RHEL6.8 (32)	libcurl.so.4	✓	✓
curl from RHEL7.0 (34)	libcurl.so.4	GLIBC_2.17	✓
convert from RHEL5.11 (23)	✓	libMagick.so.10	libMagick.so.10
convert from RHEL6.8 (28)	libMagickCore.so.5	✓	liblcms.so.1
convert from RHEL7.0 (28)	libMagickCore.so.5	libtiff.so.5	✓

common executables, `/usr/bin/curl` and `/usr/bin/convert`. Dynamically linked executables, such as `curl` and `convert`, often have dozens of shared object dependencies, each of which has its own version and further dependencies. The numbers and versions of shared object dependencies of an executable on different OSes may vary a lot, as shown in the first column of Table 5.3. Therefore, sending an executable itself to execution nodes with different OSes often does not work due to the missing of dependencies. We ran `curl` and `convert` from RHEL5-7 on RHEL5-7, and collected the results in Table 5.3. When executables ran on execution nodes with different OSes, only `curl` from RHEL6 executed successfully on RHEL7, all the other attempts failed. The first incompatible dependencies for all the failed attempts are shown in Table 5.3. However, the first incompatible dependencies are often not the only incompatible dependencies. The incompatible execution environment problem gets worse for real scientific workflows.

- **Hidden Network Dependencies.** Scientific workflows often have some data dependencies from third-party websites, which are obtained by `curl`-based or `wget`-based tasks, such as `T1` shown in Figure 5.2. When the workflow is tiny, these dependencies can be easily tracked. However, when the task number of a scientific workflow reaches 100s or 1000s, these network dependencies will be buried in the middle of the huge workflow specification and difficult to track. This may cause workflow decay if some fragile network dependencies are lost before being preserved properly.

```
1        input         http://cse.research.org/blast/input
2        job.sched     http://cse.research.org/blast/job.sched  <checksum>
```

Listing 5.1: Use Dependency Map to Track Network Dependencies in a Workflow

5.3 A Framework Facilitating the Reproducibility of Scientific Workflows

Given the challenges of reproducing scientific workflows and the characteristics of scientific workflow systems, this section proposes a framework to facilitate the reproducibility of scientific workflows, which includes three parts:

- **Tracking Network Dependencies.** Instead of being hidden in the middle of scientific workflow specifications, network dependencies should be collected and tracked in a separate mechanism. For example, a file which maps data dependencies to their retrieval locations can be used to track network dependencies, as shown in Listing 5.1. This also makes it easy to redirect these dependencies if needed. Except for the name and location of a dependency, the checksum value can also be recorded in the dependency map to facilitate the integrity of data, as shown in the second line of the Listing 5.1.

- **Specifying Execution Environments for Tasks.** A task in a workflow does not simply equal to the command line from the workflow specification, it has its own requirements of the underlying execution environment, which should include at least the information about the hardware configuration (hardware architecture, CPU, memory and disk), kernel type (such as Linux and Darwin), kernel version, OS name, OS version, the root filesystem image, software dependencies, data dependencies, command line and environment variables. The hardware and kernel information can be used to select the correct execution nodes from the underlying computing resources. The root filesystem image, software, data and environment variables information can be used to create the execution environment and execute the task in it.

  Usually the tasks in a workflow share the same hardware, kernel and OS requirements. If all the tasks in a workflow also share most of their software dependencies, a single execution environment specification can be composed and used to run all the tasks. When the dependencies of different tasks vary a lot in number or size, different execution environment specifications for different tasks can be used to avoid the space and time overhead of shipping all the dependencies to every execution node and creating a huge execution environment on every execution node. To avoid composing a separate execution environment specification for every task, it is best to leave the intermediate data involved in a workflow out of execution environment specifications and allow them to be specified by the specification parser layer automatically at runtime.

- **Sending Execution Environment Specifications to Execution Nodes.** To utilize the heterogeneous resources in the computing resource layer, instead of only sending the command line of a task to an execution node, the complete execution environment specification of the task, together with the execution environment creator which can parse the specification and create the correct execution environment from it, should also be sent to the execution node.

  To prepare executing a task, the execution environment creator obtains the root filesystem image, software and data dependencies specified in an execution environment specification, creates a sandbox with sandbox techniques like virtual machines, Linux Containers and user-space tracers, and attaches the intermediate data into it. By doing this, instead of meeting the execution environment requirements of different scientific workflows on all the execution nodes, scientific workflow systems only need to guarantee the execution environment creator itself can work on every execution node. This facilitates both the portability and reproducibility of scientific workflows.

5.4 A Prototype Implementation of the Framework

A prototype of the framework was implemented by integrating *Umbrella*, an execution environment creator, into *Makeflow*, a scientific workflow system which can utilize computing resources from the local machine, SGE, Work Queue and HTCondor. To utilize Makeflow, scientists can compose their workflows in a make-similar workflow specification language, *Makefile*, as shown in Figure 5.2. In addition, a mount file (i.e., dependency map) which maps each network dependency to its retrieval location is used to track network dependencies.

Figure 5.4 shows how the framework can facilitate the reproducibility of a bioinformatics scientific workflow called *BLAST*, whose workflow specification is specified in a Makefile, *blast.makeflow*. Each task is expressed in a Makefile rule occupying two lines: the first line is in the format of `<outputs>:<inputs>`, and the second line `<cmd>`. Take the task in *blast.makeflow* as an example, the input file is `input.1`; there are three output files: `output.1`, `error.1` and `total.1`; the command is `run_blast.sh 1`. After a `makeflow` command starts, the specification parser first reads the *Makefile* file and construct a DAG representing the dependency relation-

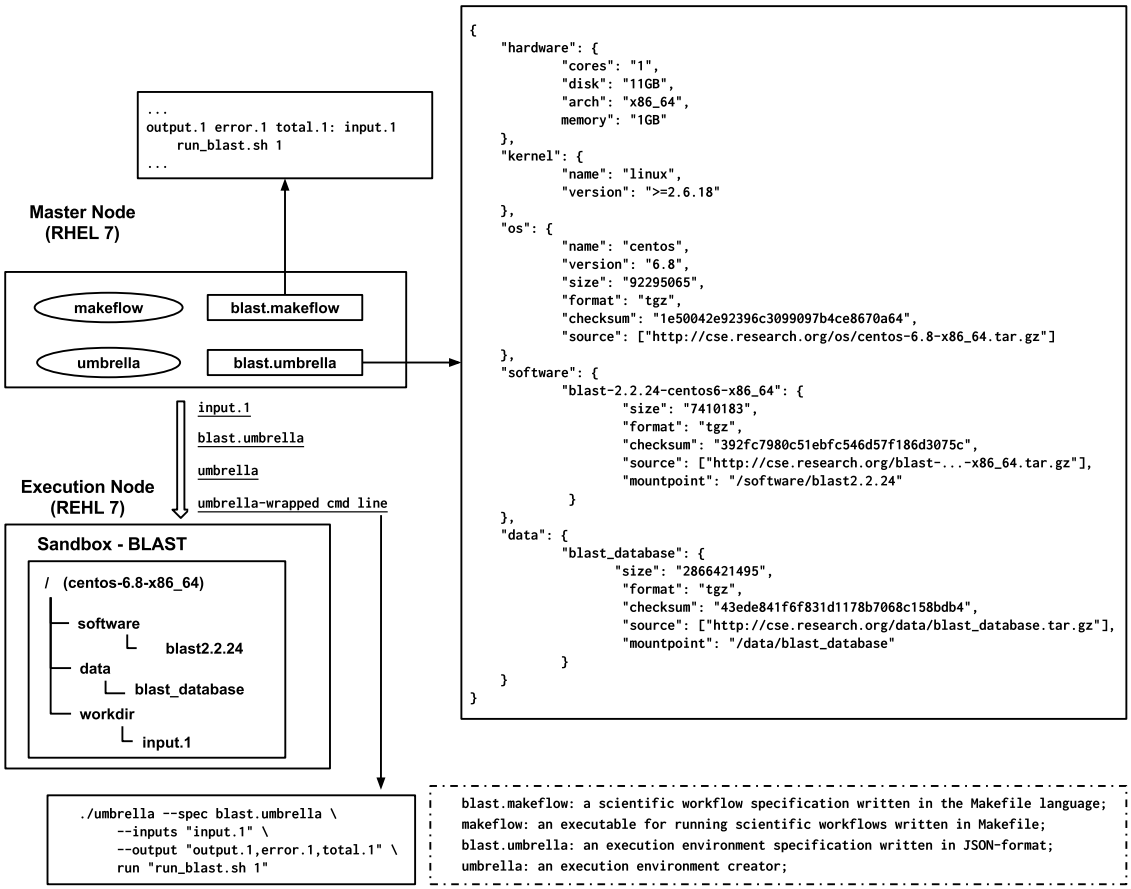

Figure 5.4. Running BLAST with Makeflow and Umbrella

ship between tasks, which can then be utilized to guide the task scheduler to schedule tasks onto execution nodes.

Without our framework, scientists need to specify the executables needed for a task, such as `run_blast.sh`, in the `inputs` part of a rule (as shown in the second line of Listing 5.2), which is cumbersome and incomplete, because these executables may depend on other libraries and executables. Using our framework, an Umbrella specification, *blast.umbrella*, is composed to specify the execution environment information including hardware, kernel, OS, software and data dependencies needed by the tasks. The workflow specification, *blast.makeflow*, can ignore the executables and focus on the inputs, outputs, and command line of each task.

```
1 ...
2 output.1 error.1 total.1: input.1 run_blast.sh
3         run_blast.sh 1
4 ...
```

Listing 5.2: Specifying Executables in Makefile – BLAST

Without our framework, the master node sends the command line of a task directly from workflow specifications to execution nodes, such as `run_blast.sh 1`, without any information about the expected execution environment. The tasks often fail on execution nodes due to incompatible execution environments. With our framework, the command line is wrapped into an Umbrella task, which brings together the execution environment specification, and the inputs, outputs and command line information from the workflow specification. The umbrella-wrapped command line, the execution environment specification, together with the execution environment creator, `umbrella`, are then sent to each execution node.

Umbrella is written in Python2.6. To run umbrella-wrapped tasks on execution nodes, the selection standards for the underlying computing resources should include the criteria for Python2.6. On execution nodes, an umbrella-wrapped task brings all the dependencies together, creates a sandbox and runs the task inside it.

5.4.1 Why not Use Docker as the Execution Environment Creator?

We tried to use Docker as the execution environment creator and run each task as a Docker container on execution nodes originally [120], and noticed the following limitations:

First, running each task as a Docker container requires Docker be installed on execution nodes and workers on execution nodes have permission to use Docker, both of which need root permission.

Second, the size limit of a Docker container (by default is 10GB) exposes a limit on the storage space used by a task. Though it is possible to tune the default container

size, the change will affect every Docker container and there is no way to set the size of each container separately.

Third, the storage drivers Docker uses usually have size limitations (e.g., 100GB used by *devicemapper* [42]). This limits the number of Docker containers which can run concurrently on an execution node.

Fourth, in the case where two different tasks pull the same Docker image or load Docker images from the same tarball, special concerns are required to solve the potential race condition.

Fifth, it is not easy to track the metadata information of software and data dependencies in a *Dockerfile*, such as size, format and checksum.

Sixth, a *Dockerfile* does not provide clear information to a new user about the location of software and data, who may want to run the Docker container with new software versions or date sets.

With Umbrella, each task can run as a Parrot job, which is a user-space ptrace-based tool and requires no root permission [75]. Umbrella does not have size limits on each task and the total storage space Umbrella can utilize is only limited by the storage limit set by a worker. Unlike Docker trying to collect all the Docker images into a centralized storage space, such as `/var/lib/docker/devicemapper`, running each task as an Umbrella job only modifies the working directory of each worker. Umbrella organizes an execution environment into different sections (such as the `software` and `data` sections in Figure 5.4), each of which has its specific purpose. This allows these dependencies to be archived separately and makes it very easy for a new user to understand the configuration of an application and test new software versions or new data sets.

5.4.2 Discussion

Integrating Umbrella into Makeflow involves changes to both sides. To support this integration, three changes were introduced to Makeflow. The first change is to allow the users of Makeflow to specify execution environment specifications for their workflows. This can be done either in workflow-level by adding a new option to Makeflow, or in task-level by adding new syntax into Makefile. The second change is to trim down Makefile rules by moving the dependencies shared by all (or a group of) tasks into execution environment specifications. However, the dependencies specific to each task should still be kept in the Makefile. The third change is to modify the task scheduler layer so that the original command line specified in a Makefile, the dependencies specific to each task, and the execution environment specification for each task can be wrapped together into an Umbrella job.

The change to Umbrella for this integration mainly is how to attach the dependencies specific to each task into an Umbrella job. Before this integration, Umbrella was mainly used to create the execution environments for single-machine scientific applications, where all the dependencies of a job is set at the same time. However, integrating Umbrella into Makeflow means there are two main sources for the dependencies of a job - an Umbrella specification for shared dependencies (such as operating systems and software) and Makefile workflow specification for the dependencies specific to each task. To achieve this, Umbrella was enhanced to allow attaching dependencies at runtime by the task scheduler layer of Makeflow.

5.5 Evaluation

Two bioinformatics scientific workflows, *BLAST* (Figure 5.5) and *BWA* (Figure 5.6), were used to evaluate our framework. The execution environments of *BLAST* and *BWA* were specified as Umbrella specification files, which were at runtime sent to

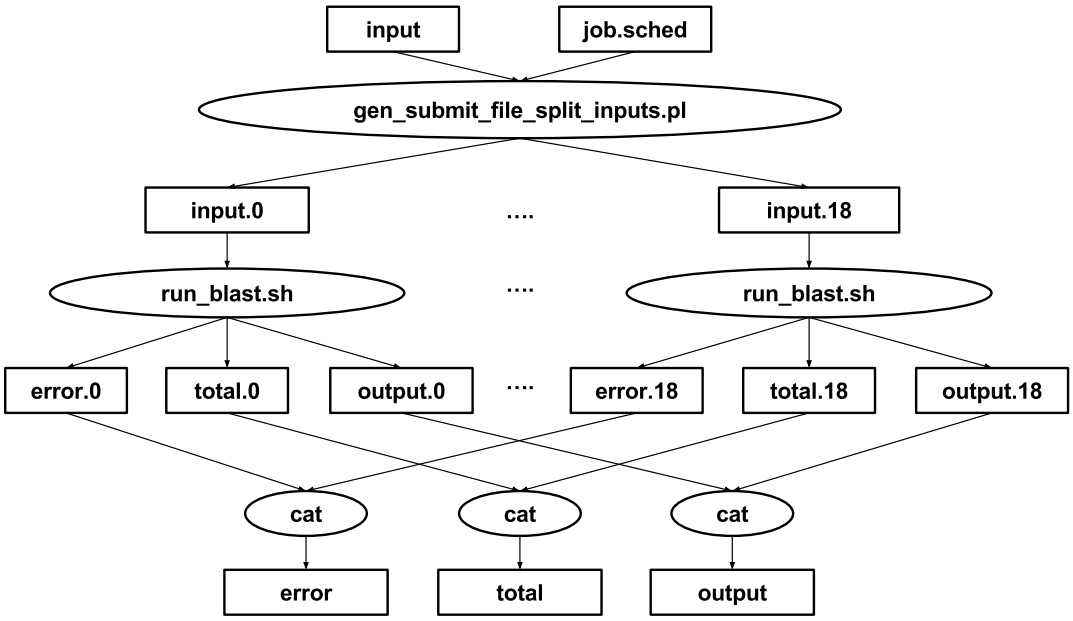

Figure 5.5. Bioinformatics Scientific Workflow - BLAST

The order matters during the concatenation of `error.*`, `total.*` *and* `output.*`, *and the correct concatenation order is from* `*.0` *all the way up to* `*.18`.

execution nodes together with the execution environment creator, `umbrella`, in the Notre Dame HTCondor pool. On each execution node, the execution environment creator creates the execution environment and run the task. The `parrot` sandbox mode of `umbrella` was used in this evaluation. For each workflow, the size of the Umbrella execution environment specification, the time and space overheads of creating execution environments using Umbrella, and the heterogeneity of execution nodes contributing to each workflow, are evaluated.

5.5.1 Applications Evaluated

BLAST (Basic Local Alignment Search Tool) is an algorithm for comparing primary biological sequence information. It allows researchers to compare a query sequence with a sequence database or library, and identify the sequences in the database or library which resemble the query sequence above a certain degree [1]. The *BLAST*

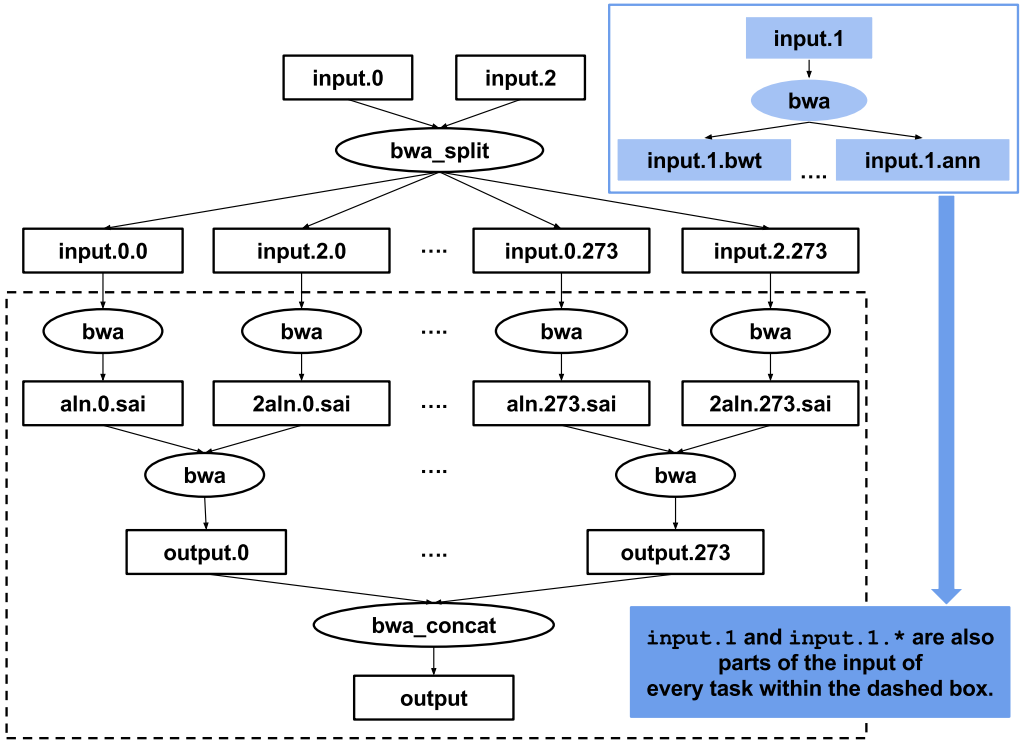

Figure 5.6. Bioinformatics Scientific Workflow - BWA

The `bwa` *program implements several efficient algorithms including* `bwa index`, `bwa aln`, `bwa sampe` *and so on. The* `bwa index` *algorithm is used to process* `input.1`*; the* `bwa aln` *algorithm is used to process* `input.0.*` *and* `input.2.*`*; the* `bwa sampe` *algorithm is used to process the* `*.sai` *files.*

program used in our evaluation takes an input file (called `input`) of 910KB including 197 query sequences, each of the query sequence is compared with a sequence database with the size of 8.4GB. The comparison result is collected into an output file with the size of 4.9MB, called `output`. Executing the whole comparison on a single machine would take about 25 hours. To speed up the execution of the comparison, a makefile is composed to specify the analysis steps (as shown in Figure 5.5), where the input file is first split into 19 tiny pieces, `input.0` - `input.18`, and then sent to multiple execution nodes. Each task takes about 60 minutes to 90 minutes. Finally, the output files returned back from the execution nodes are merged together in a certain order into a single output file.

BWA (Burrows-Wheeler Aligner) is an efficient program for mapping low-divergent sequences against a large reference genome such as the human genome [68]. It first indexes the reference genome in fasta format using the `bwa index` algorithm, then aligns the input sequence data to the reference genome using `bwa aln` algorithm. Finally the `bwa sampe` algorithm is used to generate SAM file from the alignments for paired-end reads. The *BWA* program used in our evaluation takes a fasta-format reference genome file with the size of 265MB (called `input.1`) and two input sequence files both with the size of 3.1GB (`input.0` and `input.2`) . The output SAM file (called `output`) is 8.6GB. Executing the whole program on a single machine would take days to finish. To speed up the execution of the alignment, a makefile is composed to specify the analysis steps (as shown in Figure 5.6), where each input sequence file is split into 274 tiny files and then sent to multiple execution nodes together with the index of the reference genome. Each task takes several minutes. Finally, the output files returned back from the execution nodes are merged together in a certain order into a single output file.

TABLE 5.4

OS AND SOFTWARE DEPENDENCIES OF EVALUATED WORKFLOWS

Application	OS Dependency	Software Dependencies
BLAST	CentOS 6.8 (66MB/203MB)	perl (23MB/83MB), blast (7MB/22MB)
BWA	CentOS 6.8 (66MB/203MB)	perl (23MB/83MB), bwa (216KB/604KB)

5.5.2 OS and Data Dependencies of Evaluated Applications

Except for the input and output dependencies embedded in workflow specifications, scientists can specify the underlying execution environments - including at least OS and software dependencies - for their workflows. Table 5.4 shows the OS and software dependencies of each evaluated workflow. The size information of each dependency is in the format of `tgz_format_compressed_size/uncompressed_size`. The relocatable software dependencies can be preserved and delivered as standalone packages. The non-relocatable software dependencies can be installed into the OS dependency via package managers and delivered as a fat OS dependency.

The framework allows execution environment dependencies to be archived in fine granularity such as each basic root filesystem image, each software and each dataset, which saves the space overhead of archiving dependencies. For example, both *BLAST* and *BWA* depend on the same OS image and the same `perl` binary, as shown in Table 5.4. With the framework, the OS image and the `perl` binary only need to be archived once. As the number of scientific workflows depending on the same OS image or software binary increases, archiving dependencies in fine granularity becomes more advantageous and even necessary.

TABLE 5.5

SPACE AND TIME OVERHEADS INTRODUCED BY UMBRELLA - OS AND SOFTWARE DEPENDENCIES

Application	**Umbrella Spec Size**	**Space Overhead**	**Time Overhead**
BLAST	2.2KB	404MB	<1min
BWA	1.3KB	376MB	<1min

5.5.3 Space and Time Overheads Introduced by Umbrella

The *Umbrella Spec Size* column in Table 5.5 shows the size of the Umbrella execution environment specification file for each evaluated workflow, which is very tiny and should not cause any communication problem between the master node and the execution node. The *Space Overhead* column in Table 5.5 shows the space overhead of the OS and software dependencies introduced by running each task as an Umbrella job on an execution node, which includes both the compressed and uncompressed versions of each dependency. The *Time Overhead* column shows the time overhead introduced by obtaining these dependencies, uncompressing them if necessary, and integrating them into a unified sandbox. It is worth noting that the space and time overheads shown in Table 5.5 do not include the overhead of delivering the input files and the intermediate data of a scientific workflow to execution nodes and running the tasks on execution nodes, which are necessary even without our framework.

5.5.4 Heterogeneity of Execution Nodes

The computing resources used to run the tasks for both workflows are from the Notre Dame HTCondor pool. Without our framework, the execution nodes with incompatible hardware, kernel, OS and software dependencies all cannot be utilized.

TABLE 5.6

HETEROGENEITY OF EXECUTION NODES CONTRIBUTING TO EACH WORKFLOW

Application	Total Task Number	Tasks run on RHEL6	Tasks run on RHEL7
BLAST	23	15	8
BWA	825	366	459

With our framework, *Umbrella* can be utilized to deliver the required OS and software dependencies and create the required execution environments based on the Umbrella execution environment specifications received from the master node. Therefore, tasks can run successfully on any execution node satisfying the hardware and kernel requirements. Table 5.6 shows the task number of each workflow, the number of tasks running on RHEL6 and the number of tasks running on RHEL7. Due to the preference settings on the computing resources in the HTCondor pool, all the tasks were scheduled onto the execution nodes with either RHEL6 or RHEL7. However, this does not mean these tasks can only run on RHEL6 or RHEL7.

5.6 External Reproducibility of Scientific Workflows

Integrating execution environment specifications into scientific workflow systems improves the reproducibility of scientific workflows by giving scientists complete control over the execution environment of each task in their workflows. The integration can happen in two methods. The first method is to specify an execution environment specification, which applies to every task in a workflow, as a new option to scientific workflow systems. Take the integration of Umbrella into Makeflow as an example, a new option, `--umbrella-spec`, was added into `makeflow`, as shown in the second line

```
makeflow blast.makeflow   // before the integration
makeflow --umbrella-spec blast.umbrella blast.makeflow    // after the integration
```

Listing 5.3: Specifying Execution Environment Specifications as a Command Option

```
...
.MAKEFLOW CATEGORY 1
.UMBRELLA SPEC convert.umbrella
capitol.180.jpg: capitol.jpg
        convert -swirl 180 capitol.jpg capitol.180.jpg

capitol.270.jpg: capitol.jpg
        convert -swirl 270 capitol.jpg capitol.270.jpg

capitol.360.jpg: capitol.jpg
        convert -swirl 360 capitol.jpg capitol.360.jpg

.MAKEFLOW CATEGORY 2
.UMBRELLA SPEC curl.umbrella
capitol.jpg:
        curl -o capitol.jpg http://ccl.cse.nd.edu/images/capitol.jpg
...
```

Listing 5.4: Specifying Execution Environment Specifications inside Workflow Specifications

of Listing 5.3. The second method is to add the execution environment specifications into workflow specifications. Listing 5.4 gives an example of adding Umbrella specifications into Makefile, where `convert.umbrella` specifies the execution environment for the three `convert` tasks and `curl.umbrella` specifies the execution environment for the `curl` task.

Both of these two integration methods focus on the internal reproducibility of scientific workflows and does not target the external reproducibility of scientific workflows, i.e., how to reproduce the commands to execute scientific workflows, such as `makeflow`. These commands have their requirements on the hardware, kernel, OS and so on. To preserve and reproduce the execution environment used by a scientific workflow command, we can either create a minimal execution environment package or specify the execution environment in an organized method.

To test how well the preservation approaches used to preserve and reproduce single-machine scientific applications can help reproduce scientific workflows. *Parrot*

```
parrot_run --name-list namelist1 -env-list envlist1 makeflow blast.makeflow
parrot_package_create --name-list namelist1 --env-list envlist1 --package-path /tmp/
    package_blast
parrot_package_run --package-path /tmp/package_blast makeflow blast.makeflow
```

Listing 5.5: Applying Parrot Packaging Tool on Makeflow

```
umbrella --spec makeflow.umbrella run 'makeflow blast.makeflow'
```

Listing 5.6: Applying Umbrella on Makeflow

Packaging Tool discussed in Chapter 3 was utilized to create a minimal execution environment package for executing Makeflow scientific workflow on a master node, as shown in Listing 5.5. *Umbrella* discussed in Chapter 4 was utilized to specify the execution environment required to execute Makeflow scientific workflows, as shown in the first line of Listing 5.6.

The exploration only succeeded when the `local` execution engine of Makeflow is used and failed on all the other execution engines, which all involve computing and storage resources from multiple nodes. With the `local` execution engine, all the computing and storage resources used are from the master node and can be tracked and collected by *Parrot Packaging Tool* and *Umbrella*. In the case where resources from multiple nodes are used, *Parrot Packaging Tool* and *Umbrella* can only track and collect the dependencies on the master node and cannot track the dependencies from the worker nodes. This only helps when the performance of running the workflows in the `local` execution engine is acceptable.

Another interesting exploration we did was to specify the execution environment for a scientific workflow and also specify the execution environments for each task in the workflow. In Listing 5.7, an Umbrella specification, `makeflow.umbrella`, specifies the execution environment of the `makeflow` command; another Umbrella specification, `blast.umbrella`, specifies the execution environment for each task in the makeflow, `blast.makeflow`. This exploration failed due to the unexpected behaviors

```
umbrella --spec makeflow.umbrella run 'makeflow --umbrella-spec blast.umbrella blast
    .makeflow'
```

Listing 5.7: Applying Nesting Umbrella on Makeflow

of using sandbox techniques inside each other. The sandbox techniques used to create execution environments usually are based on virtualization, which may be hardware virtualization, OS-level virtualization or application virtualization. It works to run application virtualization within hardware virtualization or OS-level virtualization, however, running hardware virtualization or OS-level virtualization within application virtualization may not always work. For example, *Parrot Packaging Tool*, as an example of application virtualization, bases on the `ptrace` system call trapping tool; *VirtualBox*, as an example of hardware virtualization, is a setuid program; running *VirtualBox* within *Parrot Packaging Tool* would fail because `ptrace` ignores the setuid bit of a setuid program. Running application virtualization inside application virtualization (e.g., running *ptrace* inside *ptrace*), running hardware virtualization inside hardware virtualization (e.g., running VMs inside VMs), running OS-level virtualization inside OS-level virtualization (e.g., running Docker inside Docker) may fail. The exploration imposes a question of whether it is really necessary to use sandbox techniques inside each other. In the case where multiple levels of sandbox techniques are needed, we should carefully think about how to stack them, which one should be the outer layer and which one should be the inner layer.

A even higher-level reproducibility of scientific workflows is about how to make it easy to run a scientific workflow on a different scientific workflow system. For example, how to run a *Makeflow* specified in the Makefile language on *Swift*, which does not understand the Makefile language at all? The barrier between different scientific workflow systems includes language specifications, task scheduling, resource management and so on. From this perspective, a script including all the transformation and analysis steps involved in a scientific workflow should also be preserved.

5.7 Conclusion

This chapter explores the challenges in reproducing scientific workflows and proposes a framework for facilitating the reproducibility of scientific workflows by giving scientists complete control over the execution environments for their workflows and integrating execution environment specifications into scientific workflow systems. Our framework allows the dependencies of execution environments to be archived separately in finer granularity instead of gigantic all-in-one images. A prototype was implemented by integrating *Umbrella*, the execution environment creator explained in Chapter 4, into *Makeflow*, a scientific workflow system which can utilize computing resources from a HTCondor cluster. We also compared *Umbrella* and *Docker* upon their roles as the execution environment creator.

Two bioinformatics scientific workflows, *BLAST* and *BWA*, were used to evaluate our framework. The execution environments of *BLAST* and *BWA* were specified as Umbrella specification files, which were at runtime sent to execution nodes together with the execution environment creator, `umbrella`, in the Notre Dame HTCondor pool. On each execution node, the execution environment creator was used to create the execution environment and run the task. For each workflow, the size of the Umbrella execution environment specification, the time and space overheads of creating execution environments using Umbrella, and the heterogeneity of execution nodes contributing to each workflow, are evaluated.

Our evaluation results suggest that by giving scientists complete control over the underlying execution environments for their scientific workflows, sysadmins can be relieved from the burden of installing and configuring execution environments for different scientific workflows on every execution node; computing resources which could not be utilized before due to lacking of the required OS, software and data dependencies become available and can be utilized now with the help of the execution environment creator shipped to each execution node from the master node. This

improves both the portability and reproducibility of scientific workflows. The space and time overheads introduced by the execution environment creator are acceptable.

The chapter also illustrates two methods of integrating execution environment specifications into scientific workflow systems using *Umbrella* and *Makeflow* as examples, discusses how the preservation methods described in Chapter 3 and Chapter 4, can be applied onto scientific workflows to improve the reproducibility of scientific workflows, and explores how to make it easier to run a workflow on different scientific workflow systems.

CHAPTER 6

CONCLUSION

6.1 Recommendations on How to Build Reproducible Scientific Applications

This work started from the problem of how to make existing scientific applications reproducible. To this end, refinements on the preservation format and the organization of analysis code were often needed, and reflection on the execution environments used for running scientific applications was required. The lessons learned throughout this exploration should be helpful for researchers to build reproducible scientific applications from the start.

Lesson 1: Capture your entire experimental procedure into a script. The procedure of conducting an experiment is often complex and includes the steps of setting environment variables, downloading, installing and configuring software, downloading data sets, and running the analysis code. Tracking all these steps via email, documents or memory is cumbersome and error-prone. In particular, it is difficult to verify the correctness of the experimental procedure. Capturing all these steps into a script makes it easy for the original author to rerun and verify its correctness and also makes it convenient to share the work with others.

Lesson 2: document your execution environments up-front. It is common currently for researchers to leave documenting the execution environments used for scientific applications until the experiment results are collected and published, or even until the project is close to the end. However, the software stack used for an application may experience updates or upgrades; some core member working on the

project may leave; another paper deadline may rush into the schedule; the team may run out of funding for documentation; or some key steps are forgotten during the last-minute rush. Documenting execution environments up-front helps in that whenever a successful execution is reached, its configurations and steps are already there.

Lesson 3: use dependency maps to track data dependencies and compose abstract scripts. It is common currently to refer to the URLs of data dependencies in the middle of analysis code. This is not a big problem when the analysis code is tiny; however, when the analysis code reaches hundreds or thousands of lines, imports other code, or is imported into other code, tracking all the data dependencies in these codes would become challenging or even impossible. Without a method to track all the data dependencies, researchers do not have a way to judge the stability of data dependencies and preserve unstable dependencies from some third-party websites, both of which would cause the analysis code to stop working. To avoid burying dependencies in the middle of analysis code, a dependency map can be utilized to track the name of each dependency and its resource location. With dependency maps, the analysis code refers to data dependencies by their names. Resource locations can be easily tracked or even redirected in the dependency map.

After a dependency map is maintained to track all the data dependencies involved in an application, it is possible to create an abstract script which refers data dependencies by their names, not by their resource locations or file paths. Using dependency maps and abstract scripts together, it is convenient to redirect data dependencies and test the analysis code on new data sets without changing the abstract script.

Lesson 4: version control your code and data. Usually the analysis code used in an experiment is undergoing constant changes due to fixing bugs or adding new features, and may have multiple releases or versions. Using version control

systems such as Git makes it easier to track the changes and tag the versions of the code. The commit id or tag name can be attached to the experiment result to track the analysis code used for the experiment.

Unlike the code used in an experiment which may experience frequent changes, the data sets are usually very stable. However, the integrity of the data sets may be broken during activities such as incomplete data movement due to disk and network limitations, or accidental modifications. Attaching the checksum of data facilitates its integrity by making it possible to recalculate and verify the checksum on the receiver's side. One option to track the checksum of each data dependency is recording the checksum value inside the dependency map.

Lesson 5: document your motivation, goal and solution. Documenting the complex execution environments for an experiment helps others reproduce the experiment. However, the things which interest the readers first are the motivation, the goal and the solution of an experiment. The larger the knowledge gap between the original author and the readers, the more important the motivation, goal and solution are. The time gap between the original execution and the reproduction also demands a thorough documentation on the motivation, goal and solution.

Lesson 6: think about how others can benefit from your work. Delving too deep into a certain research field, it is very easy to assume others also have the background and context information required to understand your work. It would be very helpful to think about how a researcher from a different research field can benefit from your work, how a researcher can reproduce your work three months later or three years later on a different machine.

6.2 Recommendations on How to Build Tools for Facilitating Reproducible Research

This work explores two different approaches for improving the reproducibility of scientific applications and also implements a prototype for each approach. The

lessons learned throughout this exploration may be helpful to build new tools for facilitating reproducible research.

Recommendation 1: differentiate preservation approaches from preservation tools. This differentiation will benefit the cases where others are interested in a preservation approach, but would like to implement their own preservation tools which may have more friendly user interface, better performance, or target different platforms. For example, the prototype implemented for the preservation approach proposed in Chapter 3, *Parrot Packaging Tool*, only works on Linux. However, the approach of tracking the accessed files and creating minimal execution environment packages is broadly applicable for other operating systems.

Recommendation 2: make the preserved artifacts easy to be interpreted by different tools. The preserved artifacts may be created by a specific preservation tool, but should be easily interpreted by other tools for the following reasons: first, the tool which was attached to the preserved artifacts originally may be out of date and out of support; second, the tool which was attached to the preserved artifacts may not work on some platforms with different operating systems; third, new tools may perform better than the original tool due to the usage of more advanced techniques; fourth, the original tool may require a special authority the user does not have. In Chapter 3, the format of the preserved artifacts created by *Parrot Packaging Tool* is a subtree of a typical Unix file system structure and can be interpreted directly by different tools like *Parrot Packaging Tool*, chroot and PTU. They can also be converted into Docker images or virtual machine images easily, and then run as Docker containers or virtual machines. This improves both portability and reproducibility.

Recommendation 3: decouple execution environment specifications from tools used to create the specified execution environments. The tools used to create execution environments may experience frequent changes due to technological evolution. However, execution environment specifications should be lightweight,

persistent and deployable. Take this work as an example, hardware virtualization (i.e., virtual machines and virtual appliances) were first utilized to create execution environments, then application virtualization such as Parrot were used, then OS-level virtualization such as Docker. It is difficult to know which other tools will become popular for the purpose of reproducibility next year or three years later. Therefore, decoupling execution environment specifications from the tools used to create the specified environments allows execution environment specifications to be utilized by different tools across a longer time range.

6.3 How to Reproduce Old and New Applications in the Future

This dissertation describes several tools which can be utilized to improve the reproducibility of scientific applications. According to their purpose, these tools can be divided into two categories: the *preserving* tools which can be used to preserve scientific applications in a certain format; and the *reproducing* tools which can be used to reproduce scientific applications with the preserved artifacts. The preserving tools and the scientific applications being preserved usually run on the same platform; while the reproducing tools may run on a different platform currently or in the future. As time goes by, the preserving tools may become obsolete, however, the preserved artifacts preserved today may be, and should be easily interpreted by possibly new reproducing tools in the future. As a software solution, each of these tools has its own dependencies and expected execution environment. It would be worthwhile to consider, in the near and far future, how to reproduce old applications with preserved artifacts and how to reproduce new applications.

If these tools keep evolving to support new OSes, new kernel or hardware in the near future, there will be no problem to utilize them to reproduce old scientific applications with preserved artifacts and new scientific applications. However, these tools will become obsolete eventually. To reproduce new applications, new tools can be

implemented using the same preservation approaches as today or some novel preservation approaches. To reproduce old applications with preserved artifacts, except for the case where the old platforms are preserved via hardware preservation, the desired platforms should be emulated somehow. Here we have multiple choices on how to emulate the target platform on the current platform. One option is to emulate the target platform directly on the current platform. Another option is to emulate an old platform on the current platform, then emulate an older platform on the emulated old platform, until the target platform is emulated. However, nesting multiple emulation techniques would not only introduce more time and space overheads, but also impose technical difficulties. It is recommended to emulate the platform required by a scientific application directly.

6.4 Summary

Sharing single-machine scientific applications without their execution environments makes it difficult for others to reproduce and verify the experiment results. This may be caused by incompatible hardware, kernel, OS and software, or missing data dependencies, or something trivial like incorrect or incomplete environment variable settings.

Sharing scientific workflows without their execution environments makes it difficult to reproduce scientific workflows, which is known as *workflow decay*. What makes it worse is that in scientific workflow systems, researchers often do not have direct control over the underlying execution environments used to run their tasks, and sysadmins need to install and configure the execution environments required for different scientific workflows on all the execution nodes.

Various attempts have been made to enhance traditional scholarly publication and make scientific results reproducible. However, these attempts often fail to cover the complete software stack used by an application (i.e., miss parts of a software

stack); or fail on huge software stacks; or just bundle all the dependencies together without making them directly executable; or are not space-efficient, because common dependencies shared by multiple applications are preserved repeatedly.

This dissertation proposes two broad approaches for improving the reproducibility of scientific applications and explore their feasibility and applicability for single-machine scientific applications and complex scientific workflows. The first approach wraps the minimal execution environment of an application into an all-in-one package, which can then be used to reproduce the application. The second approach specifies the execution environment from hardware, kernel and OS all the way up to software, data and environment variables in an organized way, preserves dependencies in the unit of basic OS image, software and data, and combines all the dependencies of an application at runtime using mounting mechanisms. A prototype was implemented for each approach - *Parrot Packaging Tool* for the first approach and *Umbrella* for the second one.

For each approach, the following three aspects are explored: what to preserve, how to preserve and how to reproduce. The time and space overheads to preserve applications, the time and space overheads to reproduce applications, the correctness of preserved artifacts are evaluated through applications from high energy physics, bioinformatics, epidemiology and scene rendering.

The evaluation results show that both approaches allow researchers to reproduce an application and verify its results with acceptable time and space overheads. However, the second approach avoids archiving shared dependencies into multiple packages repeatedly and utilizes the storage space more efficiently by allowing the archive granularity to be each OS image, each software and each data. Specifying a complex execution environment in an organized method, as suggested by the second approach, also makes it easier for others to understand the execution environment configurations of an application and extend the original work. The first approach

is not applicable to scientific applications involving multiple software stacks, which leaves the second approach to be the only choice.

This work makes its contribution by demonstrating the importance of execution environments for the reproducibility of scientific applications, differentiating execution environment specifications, which should be lightweight, persistent and deployable, from various tools used to create execution environments which may experience frequent changes due to technological evolution. It provides two prototypes to assist researchers in reproducing their applications for both the purposes of result verification and research extension, and gives recommendations on how to build reproducible scientific applications from the start.

6.5 Future Work

This work improves the reproducibility of scientific applications to some extent, but does not solve the reproducibility problem completely.

This work focuses on how to reproduce deterministic scientific applications, does not explore how to reproduce non-deterministic scientific applications. For deterministic applications, running a single application multiple times always generates the same output byte-by-byte, and a checksum value of the output can be used to verify the correctness of each run. For non-deterministic applications, random data such as timestamps may be put into the output of each run, and running the same application multiple times usually generates different outputs. In this case, it is impossible to count on a checksum value for the correctness of each run. Instead, the statistics information of each output should be utilized to verify its correctness. For example, for a non-deterministic application used to conduct event simulation, the number of events, the size and attributes of each event may be used to check the correctness.

This work focuses on output-oriented result verification, does not solve the problem of how to verify performance-oriented scientific results. In addition, this work

focuses on how to reproduce single-threading applications, and may not be applicable to multiple-threading applications. This work discusses how to reproduce single-machine scientific applications and scientific workflows, but does not solve the problem of how to reproduce software computing framework like HTCondor. This work discusses how to reproduce scientific workflows which can be described as DAGs, and may not be applicable to other scientific workflows.

The two preservation approaches proposed in this work are generic, however the two prototypes - *Parrot Packaging Tool* and *Umbrella* - are both Linux-focused, cannot be used to reproduce scientific applications running on other operating systems like Mac or Windows. In addition, currently these prototypes mainly target for professional Linux users due to their command line interfaces. To improve the easiness of using these tools, graphic user interfaces can be added. To make composing Umbrella specifications easier, the sections and attributes which require to be specified by users should be reduced to the minimum.

This preservation approaches in this work does not consider the data security problem. Researchers are still responsible for checking the potential security issue before publishing and sharing execution environment packages and execution environment specifications with others.

The matrix for quantifying the reproducibility of scientific applications used in this work is a Yes/No judgment, which may not be the best way. How to disseminate the reproducibility problem into multiple stages should be considered.

6.6 Final Words

Compared with the efforts of helping researchers use different tools to preserve and reproduce their work, the real challenging part in this work is to convince researchers of the importance of reproducibility. Except for preservation approaches and preservation tools, achieving reproducibility also requires:

- *Collaboration between researchers, sysadmins, archivists and designers of preservation tools.* Every party will benefit from this collaboration. Researchers can have a deeper understanding of the dependencies involved in their experiments. In addition, their research will get more recognized by the peer researchers due to its higher reproducibility, and sharing work between researchers will also become greatly easier. The effort of sysadmins to assist researchers to sort out their execution environments will reduce the burden of configuring these execution environments on every execution node, because the reproducing tool can be used to create execution environments using sandbox techniques. Archivists can utilize the storage space in their archives more efficiently and this collaboration will also make it easier to reference and cite data. Tool designers can focus on how to specify and create execution environments without worrying about how to archive data and software.

- *Proper academic culture encouraging, requiring and even rewarding the practice of improving the reproducibility of scientific applications.* Currently, some academic conferences and journals already started working towards this direction by requiring the authors to submit the codes used for their experiment results. What is still missing is an efficient way to allow reviewers to easily rerun the codes and verify the results. A high bar for paper acceptance will strongly motivate researchers to make their work reproducible.

- *Research habits which really agree with the importance of documenting research work.* The preservation tools have a way to force researchers to specify the parts of an execution environment which are vital for the execution of an application, such as hardware, kernel, OS, software and data. It is easy to notice and warn researchers if some of these vital parts are invalid. However, it is difficult for the preservation tools to check the validity of documents about the goal, motivation and solution description. These parts are very important for the readers to understand the background and context of an application.

Once the proper academic culture and research habits are formed, researchers may stop composing irreproducible scientific applications first and then struggling to make these applications reproducible, instead start creating reproducible scientific applications from the start.

BIBLIOGRAPHY

1. BLAST (Basic Local Alignment Search Tool). https://en.wikipedia.org/wiki/BLAST.

2. CurateND, Curating Notre Dames Research and Scholarship for Study throughout Time. https://curate.nd.edu/.

3. PyYAML. http://pyyaml.org/download/pyyaml/PyYAML-3.10.tar.gz.

4. Amazon Elastic Compute Cloud User Guide for Linux Instances. http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/CopyingAMIs.html.

5. CMS Software Framework. https://github.com/cms-sw/cmssw.

6. The Linux Foundation. https://www.linuxfoundation.org/.

7. Open Science Framework. https://osf.io/.

8. Linux ptrace() System Call. http://man7.org/linux/man-pages/man2/ptrace.2.html.

9. CMS ttH TauAnalysis. https://github.com/cms-ttH/ttH-TauAnalysis, .

10. CMS ttH TauRoast. https://github.com/cms-ttH/ttH-TauRoast, .

11. Albrecht, Michael and Donnelly, Patrick and Bui, Peter and Thain, Douglas. Makeflow: A portable abstraction for data intensive computing on clusters, clouds, and grids. In *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, page 1. ACM, 2012.

12. Antcheva, Ilka and Ballintijn, M and Bellenot, B and Biskup, M and Brun, Rene and Buncic, N and Canal, Ph and Casadei, D and Couet, O and Fine, V and others. ROOTA C++ framework for petabyte data storage, statistical analysis and visualization. *Computer Physics Communications*, 180(12):2499–2512, 2009.

13. Armbrust, Michael and Fox, Armando and Griffith, Rean and Joseph, Anthony D and Katz, Randy and Konwinski, Andy and Lee, Gunho and Patterson, David and Rabkin, Ariel and Stoica, Ion and others. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

14. Barham, Paul and Dragovic, Boris and Fraser, Keir and Hand, Steven and Harris, Tim and Ho, Alex and Neugebauer, Rolf and Pratt, Ian and Warfield, Andrew. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

15. Bechhofer, Sean and De Roure, David and Gamble, Matthew and others. Research objects: Towards exchange and reuse of digital knowledge. *Nature Precedings (2010)*, 2010. doi: 10.1038/npre.2010.4626.1.

16. Begley, C Glenn and Ellis, Lee M. Drug development: Raise standards for preclinical cancer research. *Nature*, 483(7391):531–533, 2012.

17. Berman, Fran and Fox, Geoffrey and Hey, Anthony JG. *Grid computing: making the global infrastructure a reality*, volume 2. John Wiley and sons, 2003.

18. J. Blomer, D. Berzano, P. Buncic, I. Charalampidis, G. Ganis, G. Lestaris, R. Meusel, and V. Nicolaou. Micro-CernVM: slashing the cost of building and deploying virtual machines. In *Journal of Physics: Conference Series*, volume 513, page 032009. IOP Publishing, 2014.

19. Blomer, Jakob and Buncic, Predrag and Fuhrmann, Thomas. 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.

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

21. R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab, et al. Grid'5000: a large scale and highly reconfigurable experimental grid testbed. *International Journal of High Performance Computing Applications*, 20(4):481–494, 2006.

22. Borgman, Christine L. 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.

23. Borthakur, Dhruba. HDFS architecture guide. *HADOOP APACHE PROJECT http://hadoop. apache. org/common/docs/current/hdfs design. pdf*, 2008.

24. Buncic, Predrag and Sanchez, C Aguado and Blomer, Jakob and Franco, Leandro and Harutyunian, Artem and Mato, Pere and Yao, Yushu. CernVM–a virtual software appliance for LHC applications. In *Journal of Physics: Conference Series*, volume 219, page 042003. IOP Publishing, 2010.

25. Buneman, Peter and Khanna, Sanjeev and Wang-Chiew, Tan. Why and where: A characterization of data provenance. In *Database TheoryICDT 2001*, pages 316–330. Springer, 2001.

26. Buyya, Rajkumar. High performance cluster computing. *New Jersey: Frentice*, 1999.

27. Canvel, Brice and Hiltgen, Alain and Vaudenay, Serge and Vuagnoux, Martin. Password interception in a SSL/TLS channel. In *Annual International Cryptology Conference*, pages 583–599. Springer, 2003.

28. Castagné, Michel. Consider the Source: The Value of Source Code to Digital Preservation Strategies. *SLIS Student Research Journal*, 2(2):5, 2013.

29. Cederqvist, Per and Pesch, Roland and others. *Version management with CVS*. Network Theory Ltd., 2002.

30. Chervenak, Ann and Foster, Ian and Kesselman, Carl and Salisbury, Charles and Tuecke, Steven. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *Journal of network and computer applications*, 23(3):187–200, 2000.

31. Chiang, Wei-Fan and Gopalakrishnan, Ganesh and Rakamaric, Zvonimir and Ahn, Dong H and Lee, Gregory L. Determinism and reproducibility in large-scale HPC systems. In *Workshop on Determinism and Correctness in Parallel Programming (WoDet)*, 2013.

32. Chirigati, Fernando Seabra and Shasha, Dennis and Freire, Juliana. ReproZip: Using Provenance to Support Computational Reproducibility. In *TaPP*, 2013.

33. CMS collaboration and Acosta, D and others. CMS Physics TDR. *CERN/LHCC*, 1(2006):421, 2006.

34. CMS Collaboration and others. The CMSSW Application Framework, 2006.

35. CMS collaboration and others. Observation of a new boson at a mass of 125 GeV with the CMS experiment at the LHC. *arXiv preprint arXiv:1207.7235*, 2012.

36. X. d. C. de Carnavalet and M. Mannan. Killed by Proxy: Analyzing Client-end TLS Interception Software. In *Network and Distributed System Security Symposium (NDSS 2016), San Diego, CA, USA*, 2016.

37. Deelman, Ewa and Blythe, James and Gil, Yolanda and others. Pegasus: Mapping scientific workflows onto the grid. In *Grid Computing*, pages 11–20. Springer, 2004.

38. Delaet, Thomas and Joosen, Wouter and Van Brabant, Bart. A Survey of System Configuration Tools. In *LISA*, 2010.

39. Dierks, Tim. The transport layer security (TLS) protocol version 1.2. 2008.

40. Diethelm, Kai. The limits of reproducibility in numerical simulation. *Computing in Science & Engineering*, 14(1):64–72, 2012.

41. Dorigo, Alvise and Elmer, Peter and Furano, Fabrizio and Hanushevsky, Andrew. XROOTD-A Highly scalable architecture for data access. *WSEAS Transactions on Computers*, 1(4.3), 2005.

42. Ellenberg, Lars. Drbd 9 and device-mapper: Linux block level storage replication. In *Proceedings of the 15th International Linux System Technology Conference*, 2008.

43. Feitelson, Dror G. From repeatability to reproducibility and corroboration. *ACM SIGOPS Operating Systems Review*, 49(1):3–11, 2015.

44. Fielding, Roy and Gettys, Jim and Mogul, Jeffrey and Frystyk, Henrik and Masinter, Larry and Leach, Paul and Berners-Lee, Tim. Hypertext transfer protocol–HTTP/1.1. Technical report, 1999.

45. Freire, Juliana. Making computations and publications reproducible with vistrails. *Computing in Science & Engineering*, 14(4):18–25, 2012. doi: 10.1109/MCSE.2012.76.

46. Gentleman, Robert and Lang, Duncan Temple. Statistical analyses and reproducible research. *Journal of Computational and Graphical Statistics*, pages 1–23, 2012. doi: 10.1198/106186007X178663.

47. Gil, Yolanda and Deelman, Ewa and Ellisman, Mark and others. Examining the challenges of scientific workflows. *Ieee computer*, 40(12):26–34, 2007.

48. Goble, Carole A and Bhagat, Jiten and Aleksejevs, Sergejs and others. myExperiment: a repository and social network for the sharing of bioinformatics workflows. *Nucleic acids research*, 38(suppl 2):W677–W682, 2010.

49. Goldberg, Robert P. Survey of virtual machine research. *Computer*, 7(6):34–45, 1974.

50. Gray, Jim and Liu, David T. and Nieto-Santisteban, Maria and Szalay, Alex and DeWitt, David J. and Heber, Gerd. Scientific Data Management in the Coming Decade. *SIGMOD Rec.*, 34(4):34–41, Dec. 2005. ISSN 0163-5808. doi: 10.1145/1107499.1107503. URL `http://doi.acm.org/10.1145/1107499.1107503`.

51. Guo, Philip J and Engler, Dawson R. CDE: Using System Call Interposition to Automatically Create Portable Software Packages. In *USENIX Annual Technical Conference*, 2011.

52. Hanselman, Scott E. and Pegah, Mahmoud. The Wild Wild Waste: E-Waste. In *Proceedings of the 35th Annual ACM SIGUCCS Fall Conference*, SIGUCCS '07, pages 157–162, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-634-9. doi: 10.1145/1294046.1294083. URL `http://doi.acm.org/10.1145/1294046.1294083`.

53. Hettne, Kristina M and Wolstencroft, Katherine and Belhajjame, Khalid and others. Best Practices for Workflow Design: How to Prevent Workflow Decay. In *SWAT4LS*, 2012.

54. Hickman, Kipp and Elgamal, Taher. The SSL protocol. *Netscape communications corp*, 501, 1995.

55. Holtman, Koen. CMS data grid system overview and requirements. Technical report, CERN-CMS-NOTE-2001-037, 2001.

56. Howard, John H and Kazar, Michael L and Menees, Sherri G and Nichols, David A and Satyanarayanan, Mahadev and Sidebotham, Robert N and West, Michael J. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems (TOCS)*, 6(1):51–81, 1988.

57. Howe, Bill. Virtual Appliances, Cloud Computing, and Reproducible Research. *Computing in Science Engineering*, 14(4):36–41, 2012. doi: 10.1109/MCSE.2012.62.

58. Humphreys, Paul. *Extending ourselves: Computational science, empiricism, and scientific method*. Oxford University Press, 2004.

59. Jarmoc, Jeff and Unit, DSCT. SSL/TLS interception proxies and transitive trust. *Black Hat Europe*, 2012.

60. Jeanvoine, Emmanuel and Sarzyniec, Luc and Nussbaum, Lucas. Kadeploy3: Efficient and Scalable Operating System Provisioning for Clusters. *USENIX; login:*, 38(1):38–44, 2013.

61. Jones, Christopher and Paterno, M and Kowalkowski, J and Sexton-Kennedy, L and Tanenbaum, William. The new CMS event data model and framework. *Proceedings for Computing in High-Energy Physics (CHEP06), Mumbai, India*, 13, 2006.

62. Juve, Gideon and Deelman, Ewa and Vahi, Karan and Mehta, Gaurang and Berriman, Bruce and Berman, Benjamin P and Maechling, Phil. Scientific workflow applications on Amazon EC2. In *E-Science Workshops, 2009 5th IEEE International Conference on*, pages 59–66. IEEE, 2009.

63. Krishnan, SPT and Gonzalez, Jose L Ugia. Google compute engine. In *Building Your Next Big Thing with Google Cloud Platform*, pages 53–81. Springer, 2015.

64. Krohn, Maxwell N and Efstathopoulos, Petros and Frey, Cliff and Kaashoek, M Frans and Kohler, Eddie and Mazieres, David and Morris, Robert and Osborne, Michelle and VanDeBogart, Steve and Ziegler, David. Make Least Privilege a Right (Not a Privilege). In *HotOS*, 2005.

65. Krsul, Ivan and Ganguly, Arijit and Zhang, Jian and Fortes, Jose AB and Figueiredo, Renato J. Vmplants: Providing and managing virtual machine execution environments for grid computing. In *Supercomputing, 2004. Proceedings of the ACM/IEEE SC2004 Conference*, pages 7–7. IEEE, 2004.

66. Langlois, Philippe and Nheili, Rafife and Denis, Christophe. Numerical reproducibility: Feasibility issues. In *New Technologies, Mobility and Security (NTMS), 2015 7th International Conference on*, pages 1–5. IEEE, 2015.

67. Leavitt, Neal. Is cloud computing really ready for prime time. *Growth*, 27(5): 15–20, 2009.

68. Li, Heng and Durbin, Richard. Fast and accurate short read alignment with Burrows–Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

69. Loeliger, Jon and McCullough, Matthew. *Version Control with Git: Powerful tools and techniques for collaborative software development.* " O'Reilly Media, Inc.", 2012.

70. Loope, James. *Managing Infrastructure with Puppet: Configuration Management at Scale.* " O'Reilly Media, Inc.", 2011.

71. M. D. Hildreth. Data and Software Preservation for Open Science (DASPOS). In *2014 IEEE 30th International Conference on Data Engineering Workshops*, pages 142–146, March 2014. doi: 10.1109/ICDEW.2014.6818318.

72. M. Taufer and N. Ganesan and S. Patel. GPU-Enabled Macromolecular Simulation: Challenges and Opportunities. *Computing in Science Engineering*, 15 (1):56–65, Jan 2013. ISSN 1521-9615. doi: 10.1109/MCSE.2012.42.

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

74. Matthews, Brian and McIlwrath, Brian and Giaretta, David and Conway, Esther. The significant properties of software: A study. *JISC report, March*, 2008.

75. Meng, H and Wolf, M and Ivie, P and Woodard, A and Hildreth, M and Thain, D. A case study in preserving a high energy physics application with Parrot. In *Journal of Physics: Conference Series*, volume 664, page 032022. IOP Publishing, 2015.

76. Meng, Haiyan and Kommineni, Rupa and Pham, Quan and Gardner, Robert and Malik, Tanu and Thain, Douglas. An invariant framework for conducting reproducible computational science. *Journal of Computational Science*, 9:137–142, 2015.

77. Meng, Haiyan and Thain, Douglas. Umbrella: A Portable Environment Creator for Reproducible Computing on Clusters, Clouds, and Grids. In *Proceedings of the 8th International Workshop on Virtualization Technologies in Distributed Computing*, VTDC '15, pages 23–30, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3573-7. doi: 10.1145/2755979.2755982. URL `http://doi.acm.org/10.1145/2755979.2755982`.

78. Meng, Haiyan and Thain, Douglas. Facilitating the Reproducibility of Scientific Workflows with Execution Environment Specifications. In *Proceedings of the 2017 International Conference on Computational Science (ICCS)*, 2017.

79. Meng, Haiyan and Vyushkov, Alexander and Wolf, Matthias and Woodard, Anna and Thain, Douglas. Conducting Reproducible Research with Umbrella: Tracking, Creating, and Preserving Execution Environments. In *e-Science (e-Science), 2016 IEEE 12th International Conference on*, 2016.

80. Merkel, Dirk. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal*, 2014(239):2, 2014.

81. Merkel, Dirk. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

82. Mesirov, Jill P. Computer science. Accessible reproducible research. *Science*, 327(5964):415–416, 2010. doi: 10.1126/science.1179653.

83. Oinn, Tom and Addis, Matthew and Ferris, Justin and others. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, 2004.

84. Osterweil, Leon J and Clarke, Lori A and Ellison, Aaron M. Forecast for Reproducible Data: Partly Cloudy. *Science*, 325(5948):1622–1622, 2009.

85. Palankar, Mayur R. and Iamnitchi, Adriana and Ripeanu, Matei and Garfinkel, Simson. Amazon S3 for Science Grids: A Viable Solution? In *Proceedings of the 2008 International Workshop on Data-aware Distributed Computing*, DADC '08, pages 55–64, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-154-5. doi: 10.1145/1383519.1383526. URL `http://doi.acm.org/10.1145/1383519.1383526`.

86. Paskin, Norman. Digital object identifier (DOI) system. *Encyclopedia of library and information sciences*, 3:1586–1592, 2010.

87. Peng, Roger D. Reproducible research in computational science. *Science*, 334 (6060):1226–1227, 2011.

88. Pham, Quan and Malik, Tanu and Foster, Ian T. Using Provenance for Repeatability. In *TaPP*, 2013.

89. Pham, Quan Tran. *A framework for reproducible computational research*. PhD thesis, THE UNIVERSITY OF CHICAGO, 2014.

90. Postel, Jon and Reynolds, Joyce. File transfer protocol. 1985.

91. Prinz, Florian and Schlange, Thomas and Asadullah, Khusru. Believe it or not: how much can we rely on published data on potential drug targets? *Nature reviews Drug discovery*, 10(9):712–712, 2011.

92. Proebsting, Todd and Warren, Alex M. Repeatability and Benefaction in Computer Systems Research. Technical report, 2015. URL `http://reproducibility.cs.arizona.edu/v2/RepeatabilityTR.pdf`.

93. Reed, Daniel A and Bajcsy, Ruzena and Fernandez, Manuel A and Griffiths, Jose-Marie and Mott, Randall D and Dongarra, Jack and Johnson, Chris R and Inouye, Alan S and Miner, William and Matzke, Martha K and others. Computational science: ensuring America's competitiveness. Technical report, DTIC Document, 2005.

94. Reshetova, Elena and Karhunen, Janne and Nyman, Thomas and Asokan, N. Security of OS-level virtualization technologies. In *Nordic Conference on Secure IT Systems*, pages 77–93. Springer, 2014.

95. D. Ressman and J. Valdés. Use of Cfengine for Automated, Multi-Platform Software and Patch Distribution. In *LISA*, pages 207–218, 2000.

96. Ricci, Robert and Wong, Gary and Stoller, Leigh and Webb, Kirk and Duerig, Jonathon and Downie, Keith and Hibler, Mike. Apt: A Platform for Repeatable Research in Computer Science. *ACM SIGOPS Operating Systems Review*, 49 (1):100–107, 2015.

97. Ruiz, Cristian and Richard, Olivier and Emeras, Joseph. Reproducible software appliances for experimentation. In *Testbeds and Research Infrastructure: Development of Networks and Communities*, pages 33–42. Springer, 2014.

98. Sandberg, Russel and Goldberg, David and Kleiman, Steve and Walsh, Dan and Lyon, Bob. Design and implementation of the Sun network filesystem. In *Proceedings of the Summer USENIX conference*, pages 119–130, 1985.

99. Smith, T and Maire, N and Ross, A and others. Towards a comprehensive simulation model of malaria epidemiology and control. *Parasitology*, 135(13): 1507–1516, 2008. doi: 10.1017/S0031182008000371.

100. Stodden, Victoria and McNutt, Marcia and Bailey, David H. and Deelman, Ewa and Gil, Yolanda and Hanson, Brooks and Heroux, Michael A. and Ioannidis, John P.A. and Taufer, Michela. Enhancing reproducibility for computational methods. *Science*, 354(6317):1240–1241, 2016. ISSN 0036-8075. doi: 10.1126/science.aah6168. URL `http://science.sciencemag.org/content/354/6317/1240`.

101. Szomszor, Martin and Moreau, Luc. Recording and reasoning over data provenance in web and grid services. In *On the move to meaningful Internet systems 2003: CoopIS, DOA, and ODBASE*, pages 603–620. Springer, 2003.

102. Tansley, Robert and Bass, Mick and Smith, MacKenzie. DSpace as an open archival information system: Current status and future directions. In *Research and advanced technology for digital libraries*, pages 446–460. Springer, 2003.

103. Taylor, Ian J. and Deelman, Ewa and Gannon, Dennis B. and Shields, Matthew. *Workflows for e-Science: Scientific Workflows for Grids*. Springer Publishing Company, Incorporated, 2014. ISBN 1849966192, 9781849966191.

104. Taylor, Mischa and Vargo, Seth. *Learning Chef: A Guide to Configuration Management and Automation*. " O'Reilly Media, Inc.", 2014.

105. Taylor, Ronald C. An overview of the Hadoop/MapReduce/HBase framework and its current applications in bioinformatics. *BMC bioinformatics*, 11(12):S1, 2010.

106. Thain, Douglas and Ivie, Peter and Meng, Haiyan. Techniques for Preserving Scientific Software Executions: Preserve the Mess or Encourage Cleanliness? In *Proceedings of the 12th International Conference on Digital Preservation (iPRES)*, 2015.

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

108. Thain, Douglas and Tannenbaum, Todd and Livny, Miron. Distributed computing in practice: The Condor experience. *Concurrency-Practice and Experience*, 17(2-4):323–356, 2005.

109. Tyler, David C and McNeil, Beth. Librarians and link rot: A comparative analysis with some methodological considerations. *portal: Libraries and the Academy*, 3(4):615–632, 2003.

110. Von Laszewski, Gregor and Fox, Geoffrey C and Wang, Fugang and Younge, Andrew J and Kulshrestha, Archit and Pike, Gregory G and Smith, Warren and Voeckler, Jens and Figueiredo, Renato J and Fortes, Jose and others. Design of the futuregrid experiment management framework. In *Gateway Computing Environments Workshop (GCE), 2010*, pages 1–10. IEEE, 2010.

111. Waldspurger, Carl A. Memory resource management in VMware ESX server. *ACM SIGOPS Operating Systems Review*, 36(SI):181–194, 2002. doi: 10.1145/844128.844146.

112. Watson, Jon. Virtualbox: bits and bytes masquerading as machines. *Linux Journal*, 2008(166):1, 2008.

113. Welch, Brent and Unangst, Marc and Abbasi, Zainul and Gibson, Garth A and Mueller, Brian and Small, Jason and Zelenka, Jim and Zhou, Bin. Scalable Performance of the Panasas Parallel File System. In *FAST*, volume 8, pages 1–17, 2008.

114. Wellisch, JP and Williams, C and Ashby, S. SCRAM: Software configuration and management for the LHC Computing Grid project. *arXiv preprint cs/0306014*, 2003.

115. Wilde, Michael and Hategan, Mihael and Wozniak, Justin M and others. Swift: A language for distributed parallel scripting. *Parallel Computing*, 37(9):633–652, 2011.

116. Yu, Jia and Buyya, Rajkumar. A Taxonomy of Scientific Workflow Systems for Grid Computing. *SIGMOD Rec.*, 34(3):44–49, Sept. 2005. ISSN 0163-5808. doi: 10.1145/1084805.1084814. URL `http://doi.acm.org/10.1145/1084805.1084814`.

117. Yu, Yang. *Os-level virtualization and its applications*. ProQuest, 2007.

118. Zabolitzky, John G. Preserving software: Why and how. *Iterations: An Interdisciplinary Journal of Software History*, 1(13):1–8, 2002.

119. Zhao, Jun and Gomez-Perez, Jose Manuel and Belhajjame, Khalid and others. Why workflows break - Understanding and combating decay in Taverna workflows. In *E-Science (e-Science), 2012 IEEE 8th International Conference on*, pages 1–9. IEEE, 2012.

120. Zheng, Charles and Thain, Douglas. Integrating Containers into Workflows: A Case Study Using Makeflow, Work Queue, and Docker. In *Proceedings of the 8th International Workshop on Virtualization Technologies in Distributed Computing*, VTDC '15, pages 31–38, New York, NY, USA, 2015. ISBN 978-1-4503-3573-7. doi: 10.1145/2755979.2755984. URL `http://doi.acm.org/10.1145/2755979.2755984`.

This document was prepared & typeset with LaTeX 2ε, and formatted with NDdiss2ε *classfile (v3.2013[2013/04/16]) provided by Sameer Vijay and updated by Megan Patnott.*