Conducting Reproducible Research with Umbrella: Tracking, Creating, and Preserving Execution Environments

Haiyan Meng
and Douglas Thain
Department of Computer
Science and Engineering
University of Notre Dame
Notre Dame, IN 46556, USA
Email: hmeng,dthain@nd.edu

Alexander Vyushkov
Center for Research Computing
University of Notre Dame
Notre Dame, IN 46556, USA
Email: avyushko@nd.edu

Matthias Wolf
and Anna Woodard
Department of Physics
University of Notre Dame
Notre Dame, IN 46556, USA
Email: mwolf3,awoodard@nd.edu

***Abstract*—Publishing scientific results without the detailed execution environments describing how the results were collected makes it difficult or even impossible for the reader to reproduce the work. However, the configurations of the execution environments are too complex to be described easily by authors. To solve this problem, we propose 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 Open Science Framework (OSF). The execution engine utilizes sandbox techniques like virtual machines (VMs), 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.**

We evaluate our framework by utilizing it to reproduce three scientific applications from epidemiology, scene rendering, and high energy physics. We evaluate the time and space overhead of reproducing these applications, and the effectiveness of the chosen archive unit and mounting mechanism for allowing different applications to share dependencies. Our results show that these applications can be reproduced using different sandbox techniques successfully and efficiently, even through the overhead and performance slightly vary.

I. Introduction

Computational science has accelerated research progress in a broad spectrum of fields and provided the ability to do important research *in silico*. However there have been fewer fundamental advances in our methods for sharing scientific knowledge [1]. 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 results were achieved are often superficial and imprecise. Authors may mention the platform configuration used for their experiments in the Evaluation section - 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. A 2015 study 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 codes in the paper itself. The study also showed that only the codes of 32.3% of the papers can be rebuilt within 30 minutes, the codes of another 16% of the papers can be rebuilt with extra effort, and it was difficult to rebuild the codes of the remaining 51.7% of the papers [23].

The execution environments utilized in computational research are complex, including hardware configuration, network topology, OS, software, data and environment variable settings. There are frequent updates and modifications in both software and hardware. Investigators may not even be aware of all the details of the software stack used for their experiments [24] 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.

Various attempts have been made to enhance traditional scholarly publication and make scientific results reproducible. Research Objects [2] 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 [9], [20] and reproducible papers [8] were designed to integrate the text, code, data and other auxiliary materials to make it easier to reproduce the computations. However, it mainly focuses on software and data dependencies, and does not include the hardware, kernel and OS dependencies. Virtual machines [15] and virtual appliances [11] were used to wrap up the whole

software stack of an experiment into a virtual machine image (VMI), which can then be used to reproduce the experiment. However, this method may fail when the software stack is too large [16] and is not space-efficient, because common dependencies (e.g., shared libraries) are archived into different VMIs redundantly.

In this paper, we extend our previous work Umbrella [17], and propose a framework to help the researchers track, create and preserve their execution environments. Umbrella focuses on the reconstruction of computing execution environments for the purpose of portability and scalability on clusters, clouds and grids. The framework proposed in this paper includes three parts - the Umbrella specification, execution engine and 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 (VMWare [32]), Linux containers (Docker [19]) and user-space tracers (Parrot [30]), and computing resources from the local machine and cloud computing services like Amazon EC2. The Umbrella execution engine also allows common dependencies like base OS images to be shared by different sandboxes concurrently. The Umbrella archiver allows the users to archive their execution environments into persistent storage services like Amazon S3 and OSF.

It is worth noting that an Umbrella specification says nothing about how to create the specified execution environment - where to create and which sandbox technique to use. Separating specifying execution environments from creating execution environments makes the specification generic, persistent and stable in spite of the evolution of sandbox techniques used to create execution environments.

With this framework, the original author can make an experiment reproducible by archiving its 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 during runtime. Any other researchers having the Umbrella specification, the proper access permission to the involved resources, and the Umbrella execution engine (written in Python 2.6) can easily reproduce the experiment.

We evaluate our framework by utilizing it to reproduce three applications from epidemiology, scene rendering, and high energy physics. For each application we evaluate the size of the Umbrella specification, the time and space overhead of creating execution environments using Umbrella, and the storage effectiveness of Umbrella for allowing dependencies to be shared by multiple execution environments. We also illustrate how an archival system, curateND, can be used to archive the dependencies of an application, create a DOI for the application, and provide an overview page including references to the Umbrella specification, experiment output and other auxiliary materials.

In summary, our contributions in this paper are two-fold:

- We introduce a lightweight, persistent and deployable specification to specify the execution environment of an experiment from hardware, kernel and OS all the way up to software, data and environment variables. The specification is deployable and can be used to reproduce an experiment easily. The specification is not tied to any specific sandbox technique, and is persistent in spite of the evolution of sandbox techniques used to create execution environments.
- Instead of storing a whole software stack of an experiment including OS, software and data as a single piece, Umbrella expects the archive unit of preserved dependencies to be basic OS image, software and data, and combines all the dependencies of an experiment at runtime using mounting mechanisms. This saves the storage space of both the archival system and the computing node by allowing different experiments to share a single copy of common dependencies.

In this paper we focus on improving the reproducibility of single-machine applications. The framework proposed here is not applicable to distributed applications, which often involve multiple software stacks and the communications between them. However, we plan to extend our framework in the future work to facilitate reproducing distributed applications.

Our framework aims to reproduce an experiment and get the same output, may not be applicable to reproduce performance-focused applications for two reasons: first, the CPU and memory fields in an Umbrella specification specify the minimal requirements, not the exact requirements; second, the Umbrella specification does not cover the parameters vital to performance-focused applications, such as CPU frequency, disk speed, and network performance.

II. Why is it so difficult to reproduce experiment results?

To understand why it is so difficult to reproduce the experiment results published in an academic paper, let us examine the typical workflow of scientific research.

The computing resources used by most research institutions are maintained by professional system administrators, who install and upgrade the OS and system-level software and manage the user accounts. As a researcher, when Alice joined her new research group, she was given access to the computing resource, let us say, a server with the hostname of `lab01.phy.research.org`. As a non-root user, Alice installed software provided by some software community 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 of how to reproduce the experiment published in her paper. Telling Bob that she ran the experiment on `lab01.phy.research.org` does not help anything, because it would be unrealistic to give the access 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. `sim_sort` is software 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.

Poor Alice! Can we do something to help her?

III. A Framework for Conducting Reproducible Research

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, we propose a framework to help scientific researchers improve the reproducibility of their research. Our framework achieves this through three mechanisms:

- allows the user to specify all the necessary details about a comprehensive execution environment - from the hardware all the way up to software and data (section III-A);

```
{
  "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" ]
  }
}
```

Fig. 1. Umbrella Specification Example - povray.umbrella

- creates the specified execution environment using sandbox techniques like VMs, Linux containers and user-space tracers (section III-B);
- helps the user archive important data and software dependencies in the first place (section III-C).

A. 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 execution 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. 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

TABLE I
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

of the required OS image, which includes the basic root filesystem except 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 generated experiment results. Figure 1 illustrates the Umbrella specification for a Povray ray tracing application.

The `os` section and each subsection under the `software` and `data` sections provide detailed information about the resource location (`source`), the fixity of the resource (`checksum, size, uncompressed_size`), how to use the resource (`format, action`), and the mountpoint at runtime (`mountpoint`). 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 of each dependency. The resources may come from local filesystems, public web servers, Amazon S3, OSF, Git Repositories, or CernVM File System (CVMFS) [3] (Table I). Resources from local filesystems can be utilized directly. Resources from public web servers are downloaded via HTTP or HTTPS protocol. URLs for resources from Amazon S3, OSF and Git Repositories, may have public or private access permissions, and 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 the FUSE module or a user-space mount toolkit, Parrot [30].

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 author of an Umbrella specification may add new sections, new subsections or new attributes according to his own needs. For example, an `ec2` attribute is added to specify an VMI from Amazon EC2 within the `os` section of Figure 1.

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 case 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 implement a web portal (`http://umbrella.basicuserinterface.com/`) which allows the user to specify an execution environment by filling a form online and automatically compute 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 responds to check the semantic errors within a specification before creating the specified execution environment. In addition, Umbrella specifications are often the collaborative outcome between the researchers and system administrators. System administrators focus on the configurations of hardware, kernel and base OS image. The researchers can focus on the software, data and all the other experiment-specific configurations.

B. Creating Execution Environment: Umbrella Execution Engine

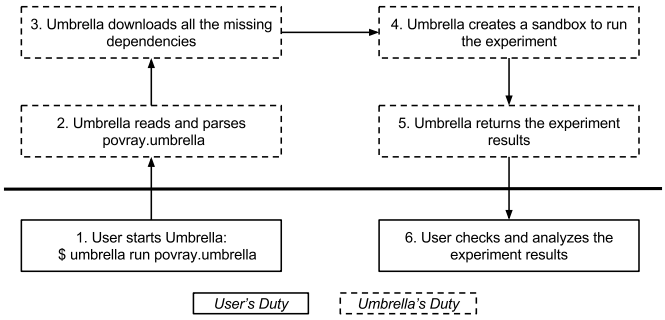

Fig. 2. Workflow of Umbrella Execution Engine

Figure 2 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 file system to run the experiment (step 4), 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 specification and the hardware, kernel and OS configurations of an execution node, different sandbox techniques can be used to create execution environments. The higher the matching

TABLE II
SANDBOX TECHNIQUES FOR CREATING EXECUTION ENVIRONMENTS

Hardware	Kernel	OS	Sandbox Techniques
Yes	Yes	Yes	Utilize the current OS directly (section III-B1)
Yes	Yes	No	OS-level Virtualization - Docker, Parrot (section III-B2)
Yes/No	No	No	Hardware Virtualization - VirtualBox, VMWare, EC2 (section III-B3)

degree is, the lighter-weight the employed sandbox techniques can be. Table II shows the available sandbox techniques for each matching degree.

1) Sandbox Technique - Utilize the Current OS Directly: When the current 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 layer being involved. However, the local filesystem may be polluted in two ways. When the `mountpoint` of a dependency has not existed yet, Umbrella should create the mountpoint before putting the dependency there. If the `mountpoint` already exists, Umbrella should first check whether the existing version is correct. In case 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.

2) Sandbox Technique - OS-Level Virtualization: If the hardware and kernel configurations of the 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. The implementation of OS-level virtualization may only deploy file system isolation (such as `chroot` and Parrot) or isolate file system 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.

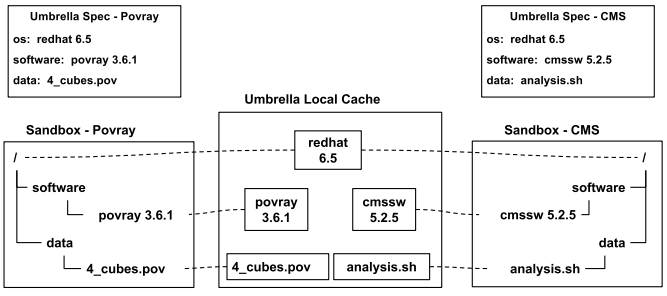

Fig. 3. Umbrella Local Cache - Allowing Dependency Sharing

This isolation also makes it possible to share the common dependencies between sandboxes for different applications, as illustrated in Figure 3. The two applications - Povray and CMS - share the same OS image (`redhat 6.5`), 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.

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 simulate the target computing environment in a virtual machine (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 the instruction on the host via a virtual machine monitor. 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 III-B1.

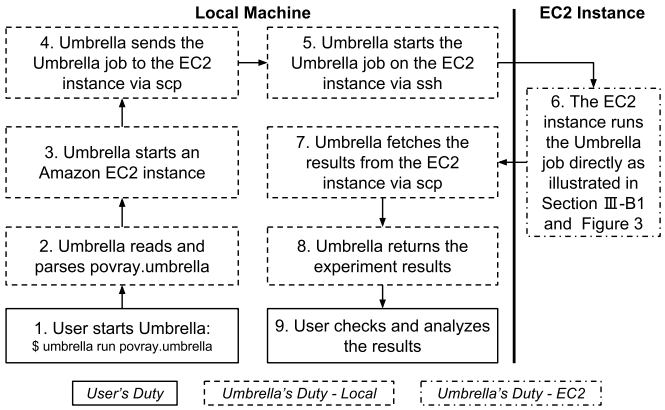

Fig. 4. Workflow of Executing An Umbrella Job via Amazon EC2

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 [33] or VMWare - 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 Figure 1. Umbrella responds to communicate with the cloud platform to start a VM, send the Umbrella job to the VM, launch the Umbrella job on the VM locally. Then the VM creates the execution environment and finishes the job. Finally, the experiments results will be sent back to the local machine, and then put into the user-specified location. Figure 4 illustrates how Umbrella can utilize Amazon EC2 to finish a job.

C. Preserving Execution Environment - Umbrella Archiver

Once the researchers figure out the final experiment settings, 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 is designed to help the researchers to archive the dependencies specified in a specification into persistent storage services. It accepts the researchers' 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 dependencies specified in a 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 resource URLs of all the relevant dependencies to the reliable ones and generate a new Umbrella specification.

Currently, Umbrella supports two persistent storage service: 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 the researcher 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 the researcher can publish and share the OSF URL of the new Umbrella specification. The archiving procedure also allows the researcher to set the access permission of the uploaded OSF and S3 resources.

IV. Example Workflows

In this section, we describe two typical scenarios where Umbrella can be used to facilitate the reproducibility of scientific research, and provide the detailed workflow of how to use Umbrella to achieve this. In both cases, 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, with all the dependencies from the local machine. Alice first composes an Umbrella specification, `povray_local.umbrella` in Figure 5, to describe the execution environment of her experiment. Then Umbrella is used to create the specified execution environment and execute

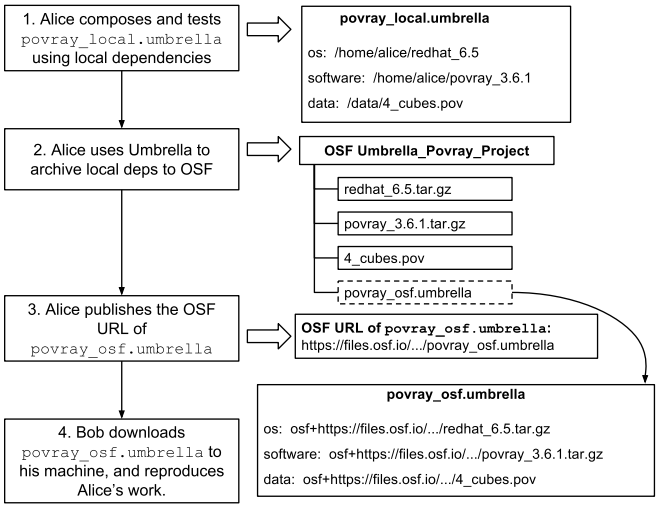

Fig. 5. Conducting Reproducible Research Using Umbrella - Local + OSF

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 of a successful execution. After Alice finishes her experiment, she uses Umbrella to creates 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. 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 5 illustrates the whole workflow.

In the second scenario, Alice wants to conduct her experiment using Amazon EC2. To avoid downloading dependencies from the outside Internet to her EC2 instance, Alice stores the software and data dependencies inside Amazon S3. Alice composes an Umbrella specification, `povray_ec2_s3.umbrella` in Figure 6, to describe the execution environment of her experiment. Then Alice uses Umbrella to communicate with AWS and executes the experiment using an EC2 instance (section III-B3). 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

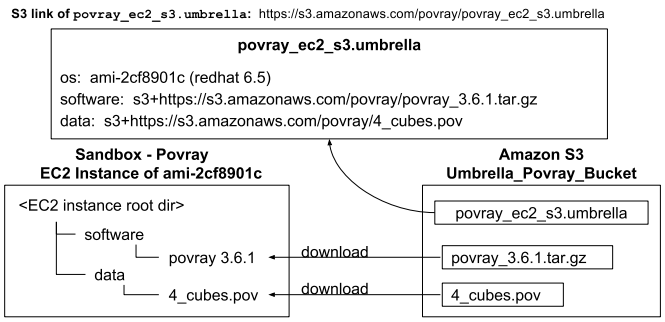

Fig. 6. Conducting Reproducible Research Using Umbrella - EC2 + S3

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.

V. Evaluation

We have implemented Umbrella supporting the features described above - tracking, creating, and preserving execution environments of scientific research - using `Python2.6`. As for creating execution environments, four sandbox techniques are currently supported by our implementation: `destructive`, which utilizes the current OS directly; `parrot` and `docker`, which are two examples of OS-level virtualization; `ec2`, which is an example of hardware virtualization utilizing Amazon EC2 (`ec2` and `destructive` are used together: the local Umbrella uses `ec2` to start an VM and the remote Umbrella running on the VM uses `destructive` to run the experiment).

We evaluate our framework by utilizing it to reproduce three scientific applications from epidemiology, scene rendering, and high energy physics (section V-A). We evaluate the Umbrella specification file sizes (section V-B), the space and time overhead of creating execution environments using different sandbox techniques (section V-C), and the effectiveness of the Umbrella local cache for allowing dependencies to be shared by multiple execution environments (section V-D). Our 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.

A. Applications Evaluated

Three applications were used to evaluate our framework: OpenMalaria from epidemiology, Povray ray tracing application, and CMS from high energy physics.

OpenMalaria aims to develop a model to simulate the potential effects of the introduction of pre-erythrocytic malaria vaccines [26]. 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`).

TABLE III
Sizes of Application Dependencies

Application	OS Deps	Software Deps	Data Deps
OpenMalaria	CentOS 6.6 (69MB/218MB)	openMalaria (2.9MB/13MB) .rpm packages (209MB) epel.repo (<1KB)	.xml (28KB) .csv (<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) parrot (23MB/71MB)	.sh (<1KB)

The size info of a plain-format dependency is in the format of `(size)`. The size info of a gzipped tar file dependency is in the format of `(compressed_size/uncompressed_size)`.

Povray is a ray tracing program which renders 3D graphics from text-based scene descriptions that describes 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 Compact Muon Solenoid (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 physics content of the generated output of a given configuration will always be identical.

Table III illustrates the dependencies of each application and the size of every dependency. Two archive formats are used for the preservation of these dependencies: plain (e.g., epel.repo) and gzipped tar file (e.g., CentOS 6.6). Both the compressed

TABLE IV
UMBRELLA SPECIFICATION FILE SIZES

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

and uncompressed size of a gzipped tar file dependency are listed in Table III. All the dependencies are archived in a campus archival system, curateND.

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

B. 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 IV 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 contents of any dependencies. Therefore, in contrast with the dependency sizes shown in Table III, an Umbrella specification file itself is a few kilobytes. To allow these applications to be reproduced with different sandbox techniques, the `os` section of each specification specifies an Amazon Machine Image (AMI) for the `ec2` sandbox mode and an OS image in the format of gzip tar file for the `parrot` and `docker` sandbox modes.

C. Overheads of Creating Execution Environments

Umbrella can create the execution environment specified in an Umbrella specification file with different sandbox techniques. Table V 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, they all generate the correct output.

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, the memory space to be at least 3GB. This feature makes it possible for an AWS user to reproduce an application using multiple instance types. We made this design decision because, from a long-term perspective, the aim of reproducibility is first to get the correct result, then to care about the performance.

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 III. The space overhead of the `docker` sandbox mode covers all the dependencies listed in Table III 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 Povray and CMS, 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.

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.

D. 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 VI 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.

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 the execution node and the curateND archival system are on the same campus network, and the downloading speed from curateND to the execution node is about 30MB/s.

E. Last Step to Enhance Reproducibility

Our framework described above facilitates the reproducibility of scientific research. To go further, we created a DOI for

TABLE V
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

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

TABLE VI
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
CMS - rerun	2.39GB	0	78min
Povray (2.40GB)	2.40GB	4.4MB (software & data)	64min
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

The initial size of the Umbrella local cache is 0. 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).

TABLE VII
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

each evaluated application, as shown in Table VII, 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.

VI. RELATED WORK

In general, there are two approaches of preserving scientific software executions: preserving the mess and encouraging cleanliness [29].

Preserving the mess allows an application to be preserved, distributed, and shared in a self-contained package. Disk cloning [13] and virtual machine image [4], [14] are two common techniques to achieve this, which are easy to reuse, but are less feasible for complex applications with large software stacks with the size of TB level. To decrease the space overhead of preserved applications, Parrot-package toolkit [18], CDE [10], PTU [22] and ReproZip [6] trap the system calls of an application and only preserve the actually used files. However, these solutions focus only on the local dependencies and do not preserve any network dependency. Furthermore, there are two main drawbacks of *preserving the mess*. First, the context and structure of the preserved applications are not usually clear enough for the new user to repurpose the application, therefore limiting their potential usage. Second, shared dependencies are redundantly preserved into multiple packages, which wastes the storage space of archival systems.

Encouraging cleanliness tries to specify an execution environment in a structured and cleanly way, archive dependencies of an application separately, and recreate the execution environment by mounting all the dependencies together into a unified sandbox. Software configuration management systems [7], like Puppet [31], Chef [27] and CFEngine [5], 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 [28]. Research Objects [2] can aggregate the data, methods and people involved in an experiment to facilitate the reproducibility of scientific results. However, Research Objects only bundle together all the necessary resource references, but are not deployable. Dynamic documents [9], [20] and reproducible papers [8] were designed to integrate the text, code, data and other auxiliary materials to make it easier to reproduce the computations. However, it mainly focuses on software and data dependencies, and does not include the hardware, kernel and OS dependencies.

Our work in this paper focuses on how to reproduce a single task. Researches focusing on improving the reproducibility of Scientific workflows, such as Prune [12], can utilize the framework proposed in this paper to reproduce each single

task in a scientific workflow.

Researches on data provenance [25] and semantic web [21] have been active to facilitate the management and reuse of scientific data. Our work focuses on how to specify and create the comprehensive execution environments of scientific applications. We have started working together with data archive services, such as curateND and OSF, to further improve the reproducibility of scientific applications.

VII. Conclusion and Future Work

In this paper, we propose a framework facilitating the conduct of reproducible research by tracking, creating and preserving the comprehensive execution environments for their experiments with Umbrella. Using this framework, the 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. The 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.

In the future work, we plan to explore the challenges involved in reproducing distributed applications, especially scientific workflows, compare different preservation degrees of workflow management systems, and extend our framework to facilitate the reproducibility of scientific workflows.

Acknowledgments

This work was supported in part by National Science Foundation grants PHY-1247316 (DASPOS), OCI-1148330 (SI2) and PHY-1312842. The University of Notre Dame Center for Research Computing scientists provided critical technical assistance throughout this research effort.

References

[1] S. Bechhofer, J. Ainsworth, J. Bhagat, et al. Why Linked Data is Not Enough for Scientists. In *e-Science (e-Science), 2010 IEEE Sixth International Conference on*, pages 300–307. IEEE, 2010.

[2] S. Bechhofer, D. De Roure, M. Gamble, et al. Research objects: Towards exchange and reuse of digital knowledge. *Nature Precedings (2010)*, 2010.

[3] J. Blomer, P. Buncic, and T. Fuhrmann. CernVM-FS: delivering scientific software to globally distributed computing resources. In *Proceedings of the first international workshop on Network-aware data management*, pages 49–56, New York, NY, USA, 2011. ACM.

[4] P. Buncic, C. A. Sanchez, J. Blomer, et al. CernVM–a virtual software appliance for LHC applications. In *Journal of Physics: Conference Series*, volume 219, page 042003. IOP Publishing, 2010.

[5] M. Burgess and O. College. Cfengine: a site configuration engine. *USENIX Computing systems*, 8(3):309–337, 1995.

[6] F. Chirigati, D. Shasha, and J. Freire. ReproZip: Using Provenance to Support Computational Reproducibility. In *Proceedings of the 5th USENIX Workshop on the Theory and Practice of Provenance*, Berkeley, CA, 2013. USENIX Association.

[7] T. Delaet, W. Joosen, and B. Van Brabant. A Survey of System Configuration Tools. In *Proceedings of the 24th International Conference on Large Installation System Administration*, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[8] J. Freire. Making computations and publications reproducible with vistrails. *Computing in Science & Engineering*, 14(4):18–25, 2012.

[9] R. Gentleman and D. T. Lang. Statistical analyses and reproducible research. *Journal of Computational and Graphical Statistics*, pages 1–23, 2012.

[10] P. J. Guo and D. R. Engler. CDE: Using System Call Interposition to Automatically Create Portable Software Packages. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, pages 21–21, Berkeley, CA, USA, 2011. USENIX Association.

[11] B. Howe. Virtual Appliances, Cloud Computing, and Reproducible Research. *Computing in Science Engineering*, 14(4):36–41, 2012.

[12] P. Ivie and D. Thain. Prune: A preserving run environment for reproducible scientific computing. In *e-Science (e-Science), 2016 IEEE 12th International Conference on*, 2016.

[13] E. Jeanvoine, L. Sarzyniec, and L. Nussbaum. Kadeploy3: Efficient and Scalable Operating System Provisioning for Clusters. *USENIX; login:*, 38(1):38–44, 2013.

[14] G. Juve, E. Deelman, K. Vahi, et al. Scientific workflow applications on Amazon EC2. In *E-Science Workshops, 2009 5th IEEE International Conference on*, pages 59–66. IEEE, 2009.

[15] I. Krsul, A. Ganguly, J. Zhang, et al. 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.

[16] H. Meng, R. Kommineni, Q. Pham, et al. An invariant framework for conducting reproducible computational science. *Journal of Computational Science*, 9:137–142, 2015.

[17] H. Meng and D. Thain. 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*, pages 23–30. ACM, 2015.

[18] H. Meng, M. Wolf, P. Ivie, et al. A case study in preserving a high energy physics application with Parrot. volume 664, page 032022. IOP Publishing, 2015.

[19] D. Merkel. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal*, 2014(239):2, 2014.

[20] J. P. Mesirov. Computer science. Accessible reproducible research. *Science*, 327(5964):415–416, 2010.

[21] J. Myers, M. Hedstrom, D. Akmon, et al. Towards sustainable curation and preservation: The sead project's data services approach. In *e-Science (e-Science), 2015 IEEE 11th International Conference on*, pages 485–494, Aug 2015.

[22] Q. T. Pham. *A framework for reproducible computational research*. PhD thesis, The University of Chicago, 2014.

[23] T. Proebsting and A. M. Warren. Repeatability and Benefaction in Computer Systems Research. Technical report, 2015.

[24] C. Ruiz, O. Richard, and J. Emeras. *Reproducible software appliances for experimentation*, pages 33–42. Springer, 2014.

[25] Y. L. Simmhan, B. Plale, and D. Gannon. A survey of data provenance in e-science. *SIGMOD Rec.*, 34(3):31–36, Sept. 2005.

[26] T. Smith, N. Maire, A. Ross, et al. Towards a comprehensive simulation model of malaria epidemiology and control. *Parasitology*, 135(13):1507–1516, 2008.

[27] D. Spinellis. Don't Install Software by Hand. *IEEE Software*, 29(4):86–87, 2012.

[28] R. Tansley, M. Bass, and M. Smith. 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.

[29] D. Thain, P. Ivie, and H. Meng. Techniques for Preserving Scientific Software Executions: Preserve the Mess or Encourage Cleanliness? *Proceedings of the 12th International Conference on Digital Preservation (iPres 2015)*, 2015.

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

[31] S. Walberg. Automate system administration tasks with puppet. *Linux Journal*, 2008(176):5, 2008.

[32] C. A. Waldspurger. Memory resource management in VMware ESX server. *ACM SIGOPS Operating Systems Review*, 36(SI):181–194, 2002.

[33] J. Watson. Virtualbox: bits and bytes masquerading as machines. *Linux Journal*, 2008(166):1, 2008.