Umbrella: A Portable Environment Creator for Reproducible Computing on Clusters, Clouds, and Grids

Haiyan Meng and Douglas Thain
Department of Computer Science and Engineering, University of Notre Dame
{hmeng|dthain}@nd.edu

ABSTRACT

Environment configuration is a significant challenge in large scale computing. An application that runs correctly on one carefully-prepared machine may fail completely on another machine, creating wasted effort and serious concerns about long-term reproducibility. Virtual machines and system containers provide a partial solution to this problem, in that they allow for the accurate reconstruction of an entire computing environment. However, when used directly, they have the dual problems of significant overhead and a lack of portability. To avoid this problem, we present Umbrella, a tool for specifying and materializing comprehensive execution environments from the hardware all the way up to software and data. A user simply invokes Umbrella with the desired task, and Umbrella determines the minimum mechanism necessary to run the task - direct execution, a system container, a local virtual machine, or submission to a cloud or grid environment. We present the overall design of Umbrella and demonstrate its use to precisely execute a high energy physics application across many platforms using combinations of chroot, Docker, Parrot, Condor, and Amazon EC2.

Keywords

execution environment, reproducible computing, containers, virtualization

1. INTRODUCTION

An application that runs correctly on one carefully-prepared machine may fail completely on another machine, creating wasted effort and serious reproducibility concerns. The reason for the failure may be incompatible hardware, mismatched kernel versions, different operating systems, missing software dependencies, wrong software versions, or just incorrect environment variables. When problems like this arise, the end user must spend considerable effort determining the nature of the incompatibility and applying the fix. It is often not immediately obvious if the problem is due to something simple like an incorrect environment variable, or something fundamental such as an incompatible kernel.

VTDC'15, June 15, 2015, Portland, Oregon, USA.

DOI: http://dx.doi.org/10.1145/2755979.2755982.

The problem becomes even more daunting when we consider large scale scientific computing. A user that wishes to harness hundreds to thousands of machines for a high-throughput computing job must confront this problem over and over again. If they have access to local resources, a high performance computing center, and a cloud provider, then porting and debugging must be repeated each time the user chooses a new platform, or the provider changes the underlying platform. In the most extreme case, a wide-area computing grid such as OSG [15] or a volunteer computing system such as BOINC [1] may provide a different environment on every single machine in the system.

The same problem may be seen in the light of portability across time. A given user may design an application on their desktop computer with a particular architecture, kernel, and OS distribution in 2015. One year later, it is likely that the architecture and kernel are the same, but the operating system has been updated (automatically) to a new version which may not be compatible with the application. Two years later, it is likely that the kernel and distribution have both changed. Five years later, perhaps the architecture has changed! Will the same application continue to run across all of these changes?

This problem is felt keenly in the high energy physics community, where hundreds of thousands of machines at multiple institutions over multiple years must be harnessed to carry out the necessary simulation and data analysis for the LHC. For example, a current version of the CMS [13] analysis codes requires the use of Linux kernel 2.6.32 running 64-bit Red Hat 6.5 and the CMS software stack which is distributed by CVMFS [2]. While attempting to run this application at the University of Notre Dame via Condor [18], we observed that, out of 4157 machines available, only 112 had the precise OS and kernel expected, and only a handful had the FUSE modules necessary to mount the CVMFS filesystem, leaving thousands of machines unharnessed.

Virtualization technology of various kinds can be helpful in solving this problem. Where available, whole-machine virtualization can be used to deploy a new kernel, OS, and software; container-based virtualization can be used to deploy a new OS and software on a compatible kernel; sandbox-based virtualization can be used to attach software and data to the filesystem image. However, each technology has tradeoffs in performance, required privilege, and usability, no single technology may be universally available across a heterogeneous system, nor (we expect) across long time scales. Furthermore, when the host machine already has the necessary environment, the user should not pay the penalty of virtualization.

To address this problem, we have created **Umbrella**, a tool for specifying and materializing execution environments, from the hardware all the way up to software and data. The end user gives a declarative specification of the desired execution environment, en-

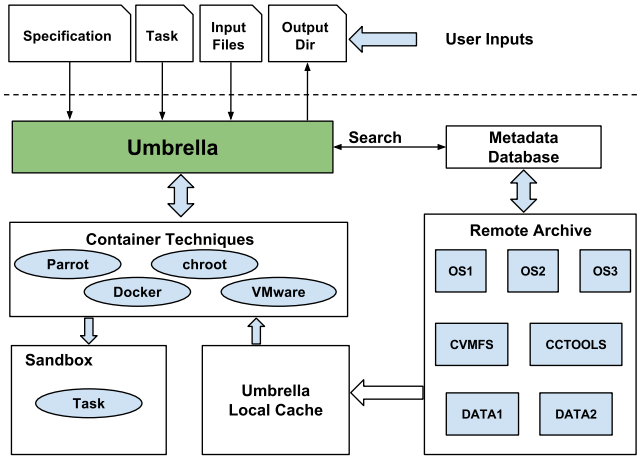


Figure 1: Architecture of Umbrella - Local Execution Engine

compassing the hardware, kernel, OS, software, data, and environment variables, without being tied down to a single virtualization technology. Umbrella accepts the execution environment specification from the user, observes the environment already present on the given machine, and provisions the desired environment using available mechanisms. In addition, Umbrella works with grid and cloud environments, using the end user's specification to select appropriate resources. Ultimately, the user's job will be executed in the desired environment, whether it is found locally on the machine, a container system constructed through sandbox techniques like Docker [14], chroot [5], Parrot [17], or a virtual machine constructed through VMware [20].

In this paper, we present the design and implementation of Umbrella. We use a CMS application to demonstrate the syntax of the environment specification, the resource matching procedure, and the materialization of the execution environment. To demonstrate that the Umbrella specification is indeed technology neutral, we show that the same application run successfully through three different sandbox technologies and two different distributed systems, observing the performance tradeoffs achieved. Finally, we use Umbrella to run thousands of tasks across our campus Condor pool, using Umbrella to shape each machine into a consistent local execution environment.

2. ARCHITECTURE

Workflow of Umbrella. To use Umbrella, the user first composes a specification to specify the execution environment for the application, and submits the specification, the task, the input files (if any), and the output directory to Umbrella. Umbrella parses the specification, checks the feasibility of the execution environment by examining the underlying execution engines. If no execution engine can provide the required environment, Umbrella suggests the user review the specification. If the specification is feasible, Umbrella figures out an execution node, determines the matching degree between the specification and the execution node, downloads the missing dependencies from the remote archive into the local cache with the help of the metadata database. Then Umbrella creates a sandbox and executes the task within it. Finally, Umbrella puts the output and results into the output directory.

Components of Umbrella Umbrella contains five parts: user inputs, Umbrella, underlying execution engines, remote archive and metadata database. User inputs include the specification, the task

```
{
    "hardware": {
        "platform": "x86_64",
        "cpu cores": "1",
        "memory": "1 GB",
        "disk": "4 GB"
    },
    "kernel": {
        "type": "linux",
        "release": ">=2.6.32"
    },
    "os": {
        "name": "RedHat",
        "version": "6.5",
        "id": "669ab5ef9....6b7907a"
    },
    "software": {
        "cmssw-5.2.5-slc5-amd64": {
            "id": "f6e17cc80….a2f4e70",
            "mountpoint": "/cvmfs/cms.cern.ch"
        }
    },
    "data": {
        "LHE_v12-ttH_samples-2381": {
            "id": "cb9878132….14be8e2",
            "mountpoint": "/tmp/2381.lhe"
        }
    },
    "environ": {
        "CMS_VERSION": "CMSSW_5_2_5"
    }
}
```

Figure 2: Specification Example - CMS Data Analysis

Umbrella allows a user to specify a dependency in two ways: unique identifier (one referent) and attribute description (a class of referents). The only except is the `environ` *section, which has a fixed syntax:* `<env_name>:<env_value>`.

command, the input files, and the output directory. Umbrella connects the user's execution environment specification with the underlying execution engines, which includes local resources, clusters, cloud resources, and grid resources. The remote archive stores the OS images, software dependencies and data dependencies. The metadata database maintains the mapping relationship between the dependency name referred in the specification and the actual storage location within the remote archive.

Local Execution Engine. Figure 1 shows the architecture of Umbrella using the local machine as the execution engine. Umbrella first determines the matching degree between the specification and the local machine. Then, the missing dependencies, which can be OS, software or data, will be downloaded into the local cache. All the dependencies specified in the specification will be grouped together and a sandbox technology (such as Parrot, chroot, or Docker) will be used to execute the task. Finally a post processing stage responds to reclaim the sandbox and ensure the output is put into the output directory specified by the user.

Specification. The specification lists all the information about the execution environment, which can be divided into six categories: hardware, kernel, OS, software, data, and environment variables. For each category, a dependency can be named by using a unique identifier, by giving the attributes of the dependency, or both. A unique identifier (`id="e5f3cd"`) makes it possible to precisely state which object in a repository must be used, but makes it more difficult to understand the intent of the user and provide alternate, compatible implementations. Attribute description of a dependency (`version="6.5"`) allows for more flexible matching, but introduces the possibility of variations between runs. The user

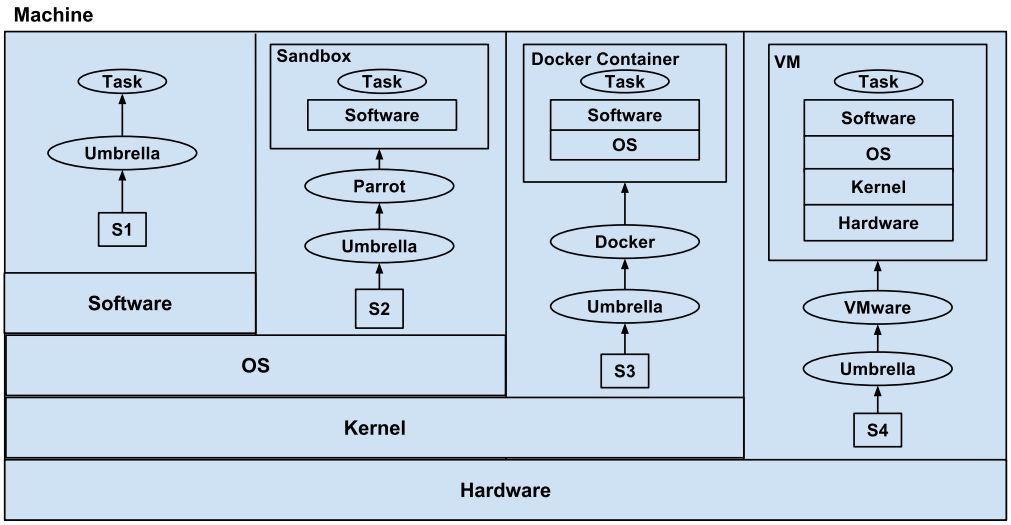


Figure 3: Umbrella Uses Varying Degrees of Virtualization

Umbrella deploys the minimum virtualization technology necessary to achieve the desired environment. (S1) If the host machine is fully compatible, the task is run directly. (S2) If the OS is compatible but some additional software or data are needed, Parrot is used to deliver the files. (S3) If only the kernel is compatible, Docker is used to deliver the operating system. (S4) If the kernel is not compatible, a virtual machine is created.

may select whichever method best meets their needs.

Figure 2 shows the specification for a CMS physics application.

The `hardware` section indicates the required CPU architecture, the CPU model, the CPU flags, the number of cores and the amount of memory, disk and other hardware requirements. The `id` attribute allows the user to specify an hardware through its unique identifier like a serial number.

The `kernel` section defines the type and version of the operating system kernel, which may be a single value or a range. The user also can specify a unique kernel image through the `id` attribute, which may be the checksum of an kernel image.

The `os` section provides the name and version information of the operating system, which includes the system software in the root filesystem, apart from the kernel. The user also can specify a unique root filesystem through the `id` attribute, which can be its checksum.

The `software` section provides the software name, version, platform of each required software package. The `id` attribute allows the user to specify a software package uniquely, which can be its checksum. The `mountpoint` attribute specifies the mount point of each software package, which will be added into the environment variables of the sandbox created for the user's task. The mountpoint attribute is the access path known to the application, and is different from the storage location of the package on the local file system, which is inside the local cache directory of Umbrella.

The `data` section indicates the necessary data dependencies, and their mount points. Similarly, the `id` attribute allows the user to specify a data package uniquely.

The `environ` section sets the environment variables for an application.

Sections of the specification may be omitted, meaning that the requirement is unknown, and Umbrella will use whatever is available.

Evaluation of Matching Degree At runtime, Umbrella evaluates the local execution environment to see if it is compatible with the specification. Umbrella evaluates the hardware resources available, the kernel and OS distribution, and the software and data dependencies. It then selects the mechanism necessary to deliver the desired environment.

The `uname` system call provides the hardware architecture and kernel information of the host machine. Python module `os` and `multiprocessing` provide the filesystem metadata including the free disk space and the CPU information of the host machine respectively. The `free` utility provides the memory usage of the host machine. If the specification provides the `id` attribute in the `os` section, Umbrella directly downloads the OS images from the archive, without checking the OS of the host machine. Otherwise, Umbrella checks the OS of the host machine with the help of `uname` and system configuration files.

The existence checking of software dependencies is difficult due to the diversity of the package sources, which may be installed by package managers, or downloaded from some websites in binary format, or even compiled from scratch on the host machine. Different settings of compilation and `PATH` environment variable make it even worse. Umbrella checks whether the required software is installed by the package manager on the local machine, and the consistency between the version installed by the package manager and the required one. If it fails, Umbrella downloads the software dependency from the remote archive.

The existence checking of data dependencies on the local machine is also difficult due to the access permission, file size, file format, and the usage of symbolic links. In addition, the data dependencies on the local machine may be changed deliberately or by accident. To use the data dependencies on the local machine means that a verification process is needed every time the dependencies are mentioned in a specification. Currently, Umbrella assumes all the data dependencies are missing on the local machine, and downloads them from the remote archive.

Figure 3 shows an example of four specifications (S1-S4) that result in different runtime deployments.

In the best case (S1), the hardware, kernel, OS, software, and even data dependencies are all ready for direct usage, Umbrella

directly executes the task on the root filesystem and execution environment of the host machine without any modification.

In the case of S2 where the hardware, kernel, and OS are all ready, and only the software and data dependencies are missing, Umbrella downloads the necessary software and data dependencies into the local cache on the local machine, sets the environment variables to make the downloaded software dependencies available, then runs the task within a sandbox. If Umbrella is run with root privileges, then a combination of `mount` and `chroot` facilities are sufficient to create a namespace for the application. If not, then Umbrella can use Parrot to intercept system calls and achieve the same effect.

In the case of S3 where only the hardware and kernel are satisfied, Umbrella downloads the OS images, together with all the software and data dependencies, into the local cache, and constructs an entirely new root filesystem containing the OS image. The application must then be run in this sandbox, using system level virtualization such as KVM [10] or Docker if the facilities are installed and available. If not, Parrot can again be used to redirect the entire filesystem image of an application, albeit at increased cost.

In the worst case (S4), even the kernel does not match the requirement, Umbrella can use hardware virtualization technologies such as VMware to create a Type II VM [6] on the host machine, then download the necessary software and data dependencies, and run the task within the VM.

Remote Archive The remote archive is a resource pool, which includes OS images for different hardware platforms, kernels, and OS distributions, software dependencies for different hardware platforms, and data dependencies. All the current available network resources can be part of the remote archive and directly used by Umbrella.

To minimize the execution environment construction time, each software dependency should be pre-built and configured. To improve the portability of archived software, software dependencies should conform to common-used internal organizations - a `bin` subdirectory for all the executables, `etc` for all the configurations, `lib` for all the libraries, `doc` for all the documents, and so on.

Local Cache One cache directory will be set on each execution node involved in the execution engine to avoid download the same data from the remote archive repeatedly. Umbrella downloads and caches OS images, software dependencies, and data dependencies in the host machine, and then creates a sandbox to execute the application. To enable software reusability by multiple users, Umbrella constructs the sandbox for each application through mounting-based sandbox techniques (Section 3). Figure 4 shows the relationship between the remote archive, the local cache and the sandbox for each application. `Sandbox 1` uses the root filesystem of the host machine as the root filesystem and mounts the needed software and data dependencies (`A` and `G`) into it. `Sandbox 2` needs to construct a separate root filesystem which groups together the needed OS image (`C`), software dependency (`A`).

Metadata Database A metadata database is set up to map the software name known to the user to the actual storage location, which will be queried by Umbrella. The metadata database also maintains the checksum and size for each archived data to verify the integrity. The granularity is set to each OS image, each software, or each dataset to avoid the storage and management overhead of metadata. Figure 5 shows the metadata for two archived software. The `source` attribute maintains a list of optional data resources to guarantee the data availability.

Umbrella separates the software and data known to the user from the delivery methods. For example, for the CMS application from Figure 2, the user specifies CMSSW as a software dependency,

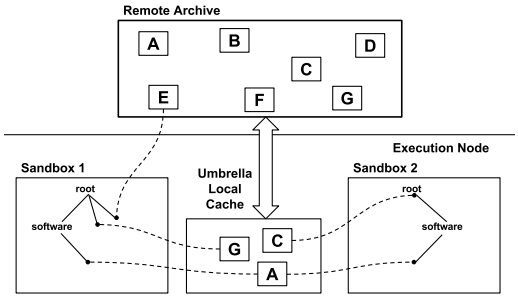


Figure 4: Mounting Mechanism

```
"cctools-4.2.2-redhat6-x86_64": {
      "source": [
            "https://.../cctools-4.2.2-redhat6-x86_64.tar.gz",
            "http://.../cctools-4.2.2-redhat6-x86_64.tar.gz"
      ],
      "checksum": "e2904442b46f3a58426c4685a82fc67b",
      "size": "8.4MB"
},
"cmssw-5.2.5-slc5-amd64": {
      "source": [
            "https://.../cmssw-5.2.5-slc5-amd64.tar.gz",
            "cvmfs://cms.cern.ch"
      ],
      "checksum": "4541620f20e4ff26faace99bffaff889",
      "size": "92MB"
}
```

Figure 5: Metadata Database

without specifying how the software should be delivered. Umbrella determines the delivery method according to the execution engine. If Parrot is chosen, the virtual filesystem feature of Parrot will be used to access CVMFS, and the delivery method will be CVMFS, which includes a repository for CMMSW. If Docker is chosen, the archived software package will be delivered to the execution node by HTTPS.

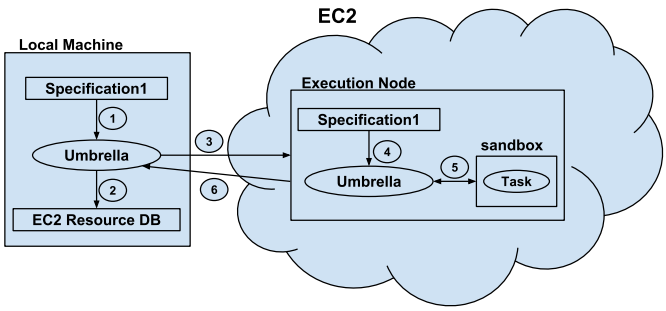
Example The command for the CMS application is shown as follows. `-T` option specifies the execution engine type, `config` option specifies the specification file, `inputs` option specifies the input files in the format of `<local-path>=<sandbox-path>`, `localdir` option specifies the directory where all the data will be cached and the sandbox will be created, `output` option specifies the output directory for the application, and the task itself is put at the end of the command inside a pair of double quotes after `run` parameter. `localdir` option is only meaningful for the local execution engine. All the input files are copied into the sandbox directory for later usage, however, the output directory is mounted into the root filesystem of the sandbox.

```
umbrella -T local --config cms.json
         --inputs '/home/1/cmd1=cms_cmd'
         --localdir /tmp
         --output /tmp/cms_output
         run "/bin/bash cms_cmd"
```

3. SANDBOX TECHNIQUES

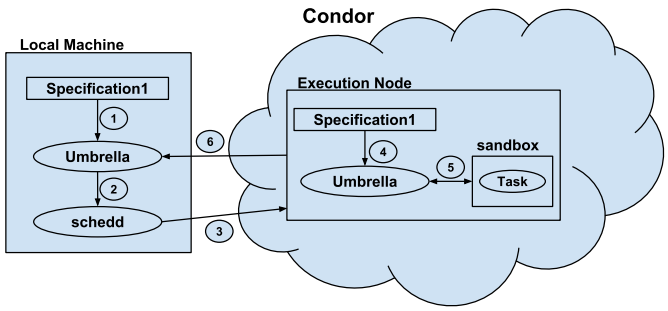
To construct a separate root filesystem combining an OS image, software and data dependencies (if any), sandbox techniques are needed. There are two different implementations of a sandbox.

One solution is to trap an application's file I/O system calls via the Linux `ptrace` debugging interface, and replacing the file ac-



1. The local machine runs Umbrella
2. Umbrella consults the EC2 Resource DB for AMI and instance type, and starts an instance
3. Umbrella sends the task to the instance
4. The instance calls Umbrella locally
5. A sandbox is created to finish the task on the instance
6. The results and output are sent back to the local machine

Figure 6: Workflow of Cloud Execution Engine - EC2



1. The local machine runs Umbrella
2. Umbrella constructs a Condor submit file, and submits the job to the schedd process of Condor
3. Condor schedules the job to one execution node
4. The execution node calls Umbrella locally
5. A sandbox is created to finish the task on the execution node
6. The results and output are sent back to the local machine

Figure 7: Workflow of Grid Execution Engines - Condor

cess path with the desired path. One example is Parrot, a virtual filesystem access tool which has been used to attach existing programs to a variety of remote I/O systems, such as HTTP, FTP, and CVMFS.

The other solution is OS-level virtualization, which allows multiple userspace instances to run on top of the same kernel concurrently. One example of this solution is `chroot`, which changes the root directory for the current process and its children by creating a chroot jail. Within a chroot jail, the process can not see the files outside the jail. Another example of this solution is `LXC`, which isolates the file system mount points and creates a separate file system layout for a group of processes. LXC uses mount namespace to create a separate root file system layout for every container, and also utilizes `cgroups` (Control Groups) to isolate and account resource usage of different process groups.

```
"redhat-6.5-x86_64": {
      "ami": "ami-1064f120",
      "root_device_type": "ebs",
      "virtualization_type": "papavirtual",
      "user": "root"
},
"instances": {
      "m1.large": {
            "cores": "4",
            "memory": "2GB",
            "disk": "20GB"
      },
        ...
}
```

Figure 8: EC2 Resource Database

4. CLOUD AND GRID INTEGRATION

As described so far, the primary job of Umbrella is to configure the local execution environment on one machine. However, the same specification of the environment also serves to *select* machines in a distributed environment. To this end, Umbrella can also submit tasks to a cloud or grid environment. In this configuration, one instance of Umbrella submits a job with appropriate requirements. The job itself consists of another instance of Umbrella which configures the local execution environment, and then runs the desired task. In the current implementation, Umbrella supports execution on Amazon EC2 and Condor.

Cloud Execution Engine - EC2 The specification for an application can be easily mapped to the EC2 resources: the hardware architecture and OS distribution information can be mapped to AMI(s), the cores, memory, disk requirements can be mapped to instance types. The AMI and EC2 instance type information are stored in a database, as shown in Figure 8. The AMI, and its root device type, virtualization type, and default user account are maintained for each OS image.

Figure 6 shows the workflow of the EC2 execution engine. Umbrella first determines the AMI and instance type according to the specification and the EC2 resource database, and starts an instance, then Umbrella sends the task, together with a copy of Umbrella, to the instance, which will run Umbrella locally to finish the task.

Grid Execution Engine - Condor The specification can also be easily mapped to the job class advertisement attributes of Condor, whereby the Umbrella command can be translated into a Condor submit file. The hardware, kernel, OS, software dependencies can be expressed in a `requirement` command, the task name can be expressed in an `executable` command, the task parameters can be expressed in an `arguments` command, the input files and output directory can be expressed in `transfer_input_files` and `transfer_output_files` commands. Once the Condor job is submitted, Umbrella gives the control to Condor, which responds to find a matched node and schedule the job to it.

Figure 7 shows the workflow of the Condor execution engine. Umbrella first checks whether the Condor execution engine has available resources to satisfy the user's requirements through the Condor APIs. If yes, Umbrella translates the requirements into a Condor submit file and then submits the job to Condor, where another instance of Umbrella runs, as in the EC2 case.

Summary Integrating cloud and grid into Umbrella makes it portable and improves the possibility of executing an application successfully. However, Umbrella does not try to change the internal principle of cloud or grid, instead it utilizes their current interfaces directly. If an execution engine takes care of job scheduling and monitoring, such as Condor, Umbrella just needs to submit a Condor job and wait for the result. If an execution engine does not support job scheduling and monitoring, like EC2, then Umbrella needs to respond for them.

Figure 9: Time Overhead of Three Sandbox Techniques

Sandbox Technique	Matching Evaluation	Software Preparation	Sandbox Construction	Application Execution	Post Processing	Total Time	Access authority
Parrot	<1s	2m 11s	<1s	5m 34s	<1s	7m 45s	any user
chroot	<1s	2m 11s	<1s	4m 33s	<1s	6m 44s	only root
Docker	<1s	2m 11s	1m 24s	4m 35s	3s	8m 13s	docker group users

Hardware Architecture: x86_64; Kernel: 3.10.0; OS: RHEL 7.0; CPUs: 4; Memory: 2GB; Free Disk: 12GB; Network: 1 Gb/s

Figure 10: Space Overhead - CMS Data Analysis

Type	Description	Size
input	specification	<1KB
input	CMS script	<1KB
os	RHEL 6.5	1.8GB
software	cmssw	327MB
software	Parrot	28MB
data	CMS event	18MB
output	ROOT file	64MB
output	analysis log	2.1MB

Figure 11: Time Overhead - Cloud Execution Engine

Subtask	Time
Start an EC2 Instance	6s
Send Task to VM	2s
Remote Execution	6m 40s
Post Processing	4s

Figure 12: Time Overhead - Grid Execution Engine

Subtask	Time
Submit File Construction	<1s
Condor Job Submission	<1s
Remote Execution	6m 20s
Post Processing	<1s

5. EVALUATION

Umbrella is written in Python 2.6 and currently uses the Condor CLI (Command Line Interface) and Amazon EC2 CLI.

We evaluated the time and space overhead of different execution engines (the local machine, Condor and Amazon EC2), and compared different sandbox techniques (Parrot, chroot and Docker) using the CMS physics application whose specification is shown in Figure 2. We showed the heterogeneity of the Notre Dame Condor Pool - hardware, kernel, OS, Linux distribution and software, ran 1000 CMS applications through the Condor Pool, and showed the distribution of the execution time and execution nodes of these applications. The CMS application takes about 5 minutes on a perfect-configured machine where all the software and data dependencies are ready.

Local Execution Engine To illustrate the time and space overhead of running applications through the local execution engine of Umbrella, we ran the CMS application on a machine which only satisfies the hardware and kernel requirements. The local execution engine downloads the missing dependencies - the OS image, software and data - from the archive, and has three options - Parrot, chroot, and Docker - to construct the sandbox to execute the application.

Figure 9 illustrates the time distribution of running the application using the three different sandbox techniques, which do the same thing during the Matching Evaluation and the Software Preparation stage and behave differently in the remaining three stages. Docker needs a longer time for the Sandbox Construction stage, because `docker import` reads and copies the OS image directory from the local cache into the docker image storage directory, which by default is `/var/lib/docker`. During the Post Processing stage, Docker needs to remove all the modifications introduced by the container and finally remove the container. Depending on a copy-on-write filesystem makes this stage longer than Parrot and chroot. Executing the application using Parrot takes longer time than Docker and chroot, because Parrot uses system call trapping via ptrace, which is slow.

Figure 10 shows the space overhead of each execution. Totally, the dependencies are 2.16GB, including OS, software, and data. The inputs include the CMS script and a specification describing the execution environment. The outputs include a ROOT file and the analysis log, totalling 66MB.

Cloud Execution Engine Figure 11 illustrates the time distribution of running the CMS application using the EC2 execution engine. Umbrella first maps the specification to EC2 AMI and instance type, then start an instance using the Amazon EC2 CLI. The following communication between Umbrella and the instance bases on SSH and SCP. During the remote execution procedure, the instance calls Umbrella locally and finishes the application through the available minimal mechanism.

Condor Execution Engine Figure 12 illustrates the time distribution of running the CMS application using the Condor execution engine. The time used to construct the submit file and submit the job into Condor is tiny. After the job is submitted, Condor takes over the job, and finds an available machine to execute the job. The configuration of the execution node affects the execution logic and execution time. The Post Processing procedure of the Condor execution engine is faster than that of the EC2 execution engine, because Condor responds to transfer the result and output back to the local machine, but Umbrella needs to actively pull the result and output from the EC2 instance.

6. UMBRELLA AT SCALE

Finally, we demonstrate the use of Umbrella to run several thousand tasks consistently on a highly heterogeneous environment. Figure 13 illustrates the heterogeneity of the Notre Dame Condor Pool - hardware, kernel, OS, Linux distribution, and software. Among the whole pool, it is difficult to find a machine where the CMS application shown in Figure 2 can run directly. The number of machines satisfying the hardware and kernel requirements is large, but the OS, software, and data requirements shrink the available machines considerably. The two virtualization technologies available on the pool are Parrot (on all machines) and Docker (on 50 machines)

Using Umbrella, we submitted 1000 different instances of the CMS application to this Condor pool, differing only in the `data` section of the specification, to control the input files. One round

Figure 13: Heterogeneity of the ND Condor Pool

Attribute	Description
machine number	4157
hardware architecture	x86_64, i386, i686
kernel version	25 kernels (2.6.18 - 3.10.0)
OS	Linux, Mac
Linux distribution	RHEL, Debian, CentOS
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	Max: 1TB, Min: 984 MB
disk size	Max: 1.7TB, Min: 5GB
docker support	50 out of 4157 have docker installed
CVMFS support	2 out of 4157 has CVMFS installed

Figure 14: Execution Time Comparison of Parrot and Docker

Case	Type	Total Time	Fastest	Slowest	Average
1	Parrot	7158m	4m 12s	11m 53s	7m 09s
2	Docker	8589m	4m 24s	13m 58s	8m 35s

of 1000 jobs was submitted, directed to use Parrot, then the same set of 1000 jobs was submitted again, directed to use Docker. We compared the output pairs in each round to ensure that both cases produced the same output.

Figure 14 shows the minimal, maximal and average execution time in both cases. Figure 15 shows the distribution of the execution time of the 1000 jobs when the execution nodes use Parrot. Due to the heterogeneity of the Condor pool and the possibility of being evicted of a running job, the execution time varies from 4 minutes to 12 minutes, which depends on the configuration of the execution node. Totally, 165 machines were utilized to finish all the jobs. Figure 16 illustrates the execution time distribution of the 1000 jobs when the execution nodes use Docker to construct sandboxes. All the jobs can finish successfully, even though about 2 minutes is needed to import the OS image into a Docker image. The portability of Docker makes more computing resources available, and makes the overhead of importing images acceptable. Totally, 25 machines were utilized to finish all the jobs.

As can be seen, Parrot and Docker have very similar minimum and maximum times, but the Parrot average was more than a minute faster. The Docker execution times also have a much greater spread in performance, despite all running on identical machines. Our best explanation is that the process of generating container images is very disk and memory intensive, and this can cause contention with other tasks running on the machine. The performance of Parrot is somewhat more compact because it is able to combine the namespaces of the underlying objects at runtime without copying or transforming images. For a longer running application, we would expect to see better runtimes for Docker, which does not have the general system call overhead of Parrot.

Regardless of the performance, the experiment demonstrates that Umbrella can effectively express and deliver the desired execution environment independently of the technology available on a given machine or a given time. Inherent in the specification is the assumption that multiple namespaces can be efficiently merged at runtime, and we believe this to be a fruitful avenue for future virtualization technologies.

7. RELATED WORK

Software configuration management systems [4] like Chef and

Figure 15: Distribution of Execution Time - Parrot

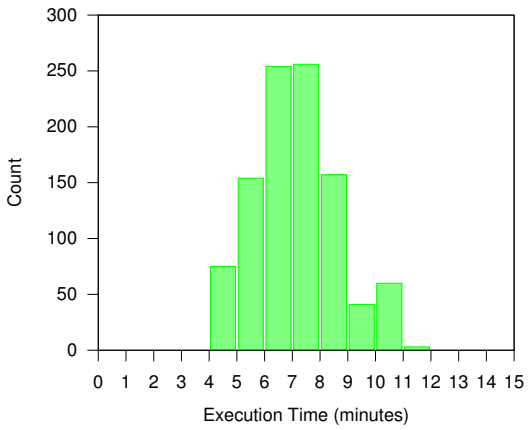
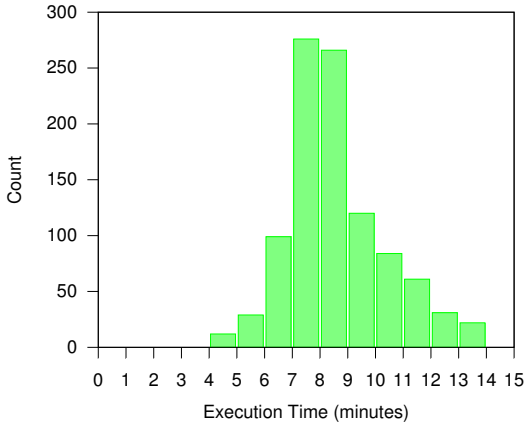


Figure 16: Distribution of Execution Time - Docker



Puppet [8] allows the users to list the application dependencies and configurations and then deploys the configurations automatically. However, software configuration management systems are designed to help system administrators to accomplish uniform and automatic deployment of configurations on multiple nodes and allows system administrators to have better control of system states.

V-MCS [16] was proposed to ease the configuration procedure of virtual machines. Execution environment management frameworks, such as FutureGrid [19], Grid'5000 [3] and VMPlants [11], were proposed to ease the execution environment configuration for grid computing. FutureGrid even allows the experiments 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. Preserving and delivering the whole software stack through virtual machines is expensive, because two virtual machine images may share a large amount of common files. In this paper, we try to explore how to allow the user to specify the execution environment through hardware, kernel, OS, software, data, and environment variables. The hierarchical specification model allows one dependency to be shared by multiple applications.

VM (Virtual Machine) [6] emulates the behavior of a particular computer system through hardware virtualization. A lot of IaaS (Infrastructure as a Service) providers, such as Amazon EC2 [9] and Microsoft Azure [12], allow their customers to use VMs for differ-

ent OSs directly without deploying them by themselves. However, a VM provided by these IaaS providers is a clean execution environment. The user must deploy the execution environment for an application before doing anything else.

Containers [21] provide multiple isolated execution instances on top of the same kernel through OS-level virtualization, such as Docker [14]. However, container techniques may require special features introduced by a later kernel version, and is not an option for applications which rely on older kernels.

Parrot [17] and CDE [7] treats software the same as data, and generates a portable package including all the files accessed by an application with the help of the `ptrace` debugging interface. However, two packages may share a lot of common file dependencies. Setting the granularity to each file also makes the metadata management and deduplication complex.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we propose Umbrella, a tool for specifying comprehensive execution environments in an organized way, from the hardware all the way up to software and data, and materializing the execution environment during runtime with the minimum mechanism, which can be local direct execution, a system container, a local VM, or submission to a cloud or grid environment. The organized Umbrella specification is light-weight, and makes an application portable and reproducible.

In the following work, we plan to optimize the organization and management of the archive to improve the scalability and the storage efficiency. Correspondingly, the retrieval interface exposed by the archive also needs to be optimized to allow the user to explore the archived packages.

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.

9. REFERENCES

[1] D. P. Anderson. BOINC: A System for Public-Resource Computing and Storage. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, GRID '04, pages 4–10, Washington, DC, USA, 2004. IEEE Computer Society.

[2] 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. ACM, 2011.

[3] 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.

[4] J. Estublier. Software configuration management: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 279–289. ACM, 2000.

[5] S. Friedl. Go directly to jail: Secure untrusted applications with chroot. *Linux Magazine*, pages 2002–12, 2002.

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

[7] P. J. Guo and D. R. Engler. CDE: Using System Call Interposition to Automatically Create Portable Software Packages. In *USENIX Annual Technical Conference*, 2011.

[8] L. Kanies. Puppet: Next-generation configuration management. *The USENIX Magazine*, 31(1):19–25, 2006.

[9] S. Khatua and N. Mukherjee. A Novel Checkpointing Scheme for Amazon EC2 Spot Instances. *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 0:180–181, 2013.

[10] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the Linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007.

[11] I. Krsul, A. Ganguly, J. Zhang, J. A. Fortes, and R. J. Figueiredo. 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.

[12] A. Lenk, M. Klems, J. Nimis, S. Tai, and T. Sandholm. What's inside the Cloud? An architectural map of the Cloud landscape. In *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, pages 23–31. IEEE Computer Society, 2009.

[13] R. McClatchey. The CMS experiment at the CERN LHC. *The Journal of Instrumentation*, 3(S08004), 2008.

[14] D. Merkel. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.*, 2014(239), Mar. 2014.

[15] R. Pordes, D. Petravick, B. Kramer, D. Olson, M. Livny, A. Roy, P. Avery, K. Blackburn, T. Wenaus, F. Würthwein, et al. The open science grid. In *Journal of Physics: Conference Series*, volume 78, page 012057. IOP Publishing, 2007.

[16] X.-H. Sun, C. Du, H. Zou, Y. Chen, and P. Shukla. V-mcs: A configuration system for virtual machines. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pages 1–7. IEEE, 2009.

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

[18] D. Thain, T. Tannenbaum, and M. Livny. Condor and the Grid. *Grid computing: Making the global infrastructure a reality*, pages 299–335, 2003.

[19] G. Von Laszewski, G. C. Fox, F. Wang, A. J. Younge, A. Kulshrestha, G. G. Pike, W. Smith, J. Voeckler, R. J. Figueiredo, J. Fortes, et al. Design of the futuregrid experiment management framework. In *Gateway computing environments workshop (GCE)*, pages 1–10, 2010.

[20] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, Dec. 2002.

[21] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. De Rose. Performance evaluation of container-based virtualization for high performance computing environments. In *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, pages 233–240. IEEE, 2013.