CDE: Run Any Linux Application On-Demand Without Installation

Philip J. Guo
Stanford University
pg@cs.stanford.edu

Abstract

There is a huge ecosystem of free software for Linux, but since each Linux distribution (distro) contains a different set of pre-installed shared libraries, filesystem layout conventions, and other environmental state, it is difficult to create and distribute software that works without hassle across all distros. Online forums and mailing lists are filled with discussions of users' troubles with compiling, installing, and configuring Linux software and their myriad of dependencies. To address this ubiquitous problem, we have created an open-source tool called CDE that automatically packages up the **C**ode, **D**ata, and **E**nvironment required to run a set of x86-Linux programs on other x86-Linux machines. Creating a CDE package is as simple as running the target application under CDE's monitoring, and executing a CDE package requires no installation, configuration, or root permissions. CDE enables Linux users to instantly run any application on-demand without encountering "dependency hell".

1 Introduction

The simple-sounding task of taking software that runs on one person's machine and getting it to run on another machine can be painfully difficult in practice. Since no two machines are identically configured, it is hard for developers to predict the exact versions of software and libraries already installed on potential users' machines and whether those conflict with the requirements of their own software. Thus, software companies devote considerable resources to creating and testing one-click installers for products like Microsoft Office, Adobe Photoshop, and Google Chrome. Similarly, open-source developers must carefully specify the proper dependencies in order to integrate their software into package management systems [4] (e.g., RPM on Linux, MacPorts on Mac OS X). Despite these efforts, online forums and mailing lists are still filled with discussions of users' troubles with compiling, installing, and configuring software and their myriad of dependencies. For example, the official Google Chrome help forum for "install/uninstall issues" has over 5800 threads.

In addition, a study of US labor statistics predicts that by 2012, 13 million American workers will do programming in their jobs, but amongst those, only 3 million will be professional software developers [24]. Thus, there are potentially millions of people who still need to get their software to run on other machines but who are unlikely to invest the effort to create one-click installers or wrestle with package managers, since their primary job is not to release production-quality software. For example:

- **System administrators** often hack together ad-hoc utilities comprised of shell scripts and custom-compiled versions of open-source software, in order to perform system monitoring and maintenance tasks. Sysadmins want to share their custom-built tools with colleagues, quickly deploy them to other machines within their organization, and "future-proof" their scripts so that they can continue functioning even as the OS inevitably gets upgraded.
- **Research scientists** often want to deploy their computational experiments to a cluster for greater performance and parallelism, but they might not have permission from the sysadmin to install the required libraries on the cluster machines. They also want to allow colleagues to run their research code in order to reproduce and extend their experiments.
- **Software prototype designers** often want clients to be able to execute their prototypes without the hassle of installing dependencies, in order to receive continual feedback throughout the design process.

In this paper, we present an open-source tool called CDE [1] that makes it easy for people of all levels of IT expertise to get their software running on other machines without the hassle of manually creating a robust

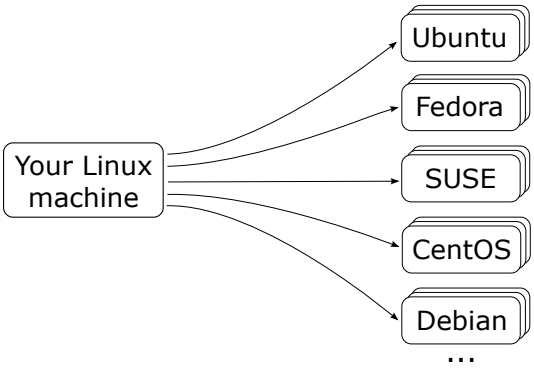

Figure 1: CDE enables users to package up any Linux application and deploy it to all modern Linux distros.

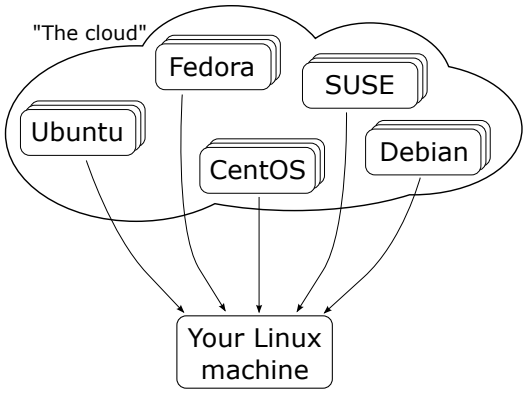

Figure 2: CDE's streaming mode enables users to run any Linux application on-demand by fetching the required files from a farm of pre-installed distros in the cloud.

installer or dealing with user complaints about dependencies. CDE automatically packages up the **C**ode, **D**ata, and **E**nvironment required to run a set of x86-Linux programs on other x86-Linux machines without any installation (see Figure 1). To use CDE, the user simply:

1. Prepends any set of Linux commands with the `cde` executable. `cde` executes the commands and uses `ptrace` system call interposition to collect all the code, data files, and environment variables used during execution into a self-contained package.
2. Copies the resulting CDE package to an x86-Linux machine running any distro from the past ∼5 years.
3. Prepends the original packaged commands with the `cde-exec` executable to run them on the target machine. `cde-exec` uses `ptrace` to redirect file-related system calls so that executables can load the required dependencies from within the package. Execution can range from ∼0% to ∼30% slower.

The main benefits of CDE are that creating a package is as easy as executing the target program under its supervision, and that running a program within a package requires no installation, configuration, or root permissions.

The design philosophy underlying CDE is that people should be able to package up their Linux software and deploy it to other Linux machines with as little effort as possible. However, CDE is not meant to replace traditional installers or package managers; its intended role is to serve as a convenient *ad-hoc* solution for people like sysadmins, research scientists, and prototype makers.

Since its release in Nov. 2010, CDE has been downloaded over 3,000 times [1]. We have exchanged hundreds of emails with users throughout both academia and industry. In the past year, we have made several significant enhancements to the base CDE system in response to user feedback. Although we introduced an early version of CDE in a short paper [20], this paper presents a more complete CDE system with three new features:

- To overcome CDE's primary limitation of only being able to package dependencies collected on executed paths, we introduce new tools and heuristics for making CDE packages complete (Section 3).
- To make CDE-packaged programs behave just like native applications on the target machine rather than executing in an isolated sandbox, we introduce a new *seamless execution mode* (Section 4).
- Finally, to enable users to run any Linux application on-demand, we introduce a new *application streaming mode* (Section 5). Figure 2 shows its high-level architecture: The system administrator first installs multiple versions of many popular Linux distros in a "distro farm" in the cloud (or an internal compute cluster). The user connects to that distro farm via an ssh-based protocol from any x86-Linux machine. The user can now run *any* application available within the package managers of any of the distros in the farm. CDE's streaming mode fetches the required files on-demand, caches them locally on the user's machine, and creates a portable distro-independent execution environment. Thus, Linux users can instantly run the hundreds of thousands of applications already available in the package managers of all distros without being forced to use one specific release of one specific distro[1].

This paper continues with descriptions of real-world use cases (Section 6), evaluations of portability and performance (Section 7), comparisons to related work (Section 8), and concludes with discussions of design philosophy, limitations, and lessons learned (Section 9).

[1]The package managers included in different releases of the same Linux distro often contain incompatible versions of many applications!

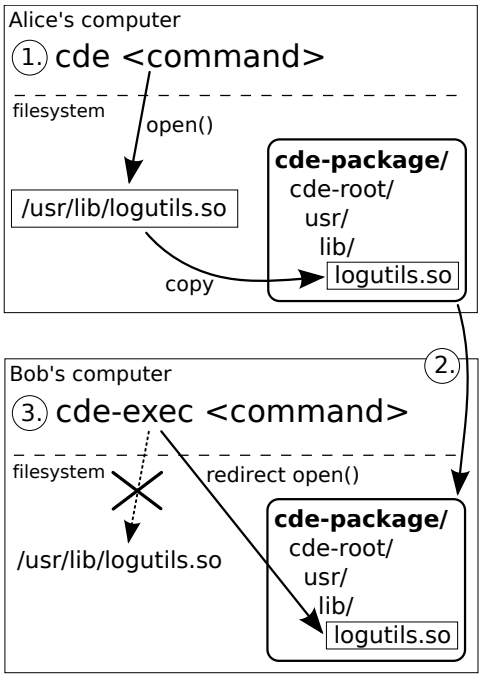

Figure 3: Example use of CDE: 1.) Alice runs her command with `cde` to create a package, 2.) Alice sends her package to Bob's computer, 3.) Bob runs command with `cde-exec`, which redirects file accesses into package.

2 CDE system overview

We described the details of CDE's design and implementation in a prior paper and its accompanying technical report [20]. We will now summarize the core features of CDE using an example.

Suppose that Alice is a system administrator who is developing a Python script to detect anomalies in network log files. She normally runs her script using this Linux command:

```
python detect_anomalies.py net.log
```

Suppose that Alice's script (`detect_anomalies.py`) imports some 3rd-party Python extension modules, which consist of optimized C++ log parsing code compiled into shared libraries. If Alice wants her colleague Bob to be able to run her analysis, then it is not sufficient to just send her script and `net.log` data file to him.

Even if Bob has a compatible version of Python on his Linux machine, he will not be able to run her script until he compiles, installs, and configures the exact extension modules that her script used (and all of their transitive dependencies). Since Bob is probably using a different Linux distribution (distro) than Alice, even if Alice precisely recalled all of the steps involved in installing all of the original dependencies on her machine, those instructions probably will not work on Bob's machine.

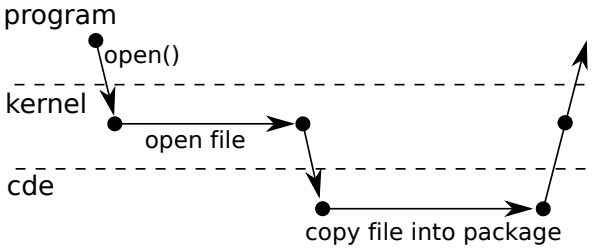

Figure 4: Timeline of control flow between target program, kernel, and `cde` process during an `open` syscall.

2.1 Creating a new CDE package

To create a self-contained package with all of the dependencies required to run her anomaly detection script on another Linux machine, Alice simply prepends her command with the `cde` executable:

```
cde python detect_anomalies.py net.log
```

`cde` runs her command normally and uses the Linux `ptrace` system call to monitor all of the files it accesses throughout execution. `cde` creates a new subdirectory called `cde-package/cde-root/` and copies all of those accessed files into there, mirroring the original directory structure. Figure 4 shows an overview of the control flow between the target program, Linux kernel, and `cde` during a file-related system call.

For example, if Alice's script dynamically loads an extension module as a shared library named `/usr/lib/logutils.so` (i.e., log parsing utility code), then `cde` will copy it to `cde-package/cde-root/usr/lib/logutils.so` (see Figure 3). `cde` also saves the values of environment variables in a text file within `cde-package/`.

When execution terminates, the `cde-package/` subdirectory (which we call a "CDE package") contains all of the files required to run Alice's original command.

2.2 Executing a CDE package

Alice zips up the `cde-package/` directory and transfers it to Bob's Linux machine. Now Bob can run Alice's anomaly detection script without first installing anything on his machine. To do so, he unzips the package, changes into the sub-directory containing the script, and prepends her original command with the `cde-exec` executable (also included in the package):

```
cde-exec python detect_anomalies.py net.log
```

`cde-exec` sets up the environment variables saved from Alice's machine and executes the versions of `python` and its extension modules that are *located within the package*. `cde-exec` uses `ptrace` to monitor all

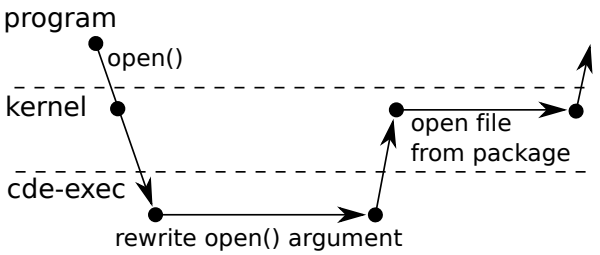

Figure 5: Timeline of control flow between target program, kernel, and `cde-exec` during an `open` syscall.

system calls that access files and dynamically rewrites their path arguments to the corresponding paths within the `cde-package/cde-root/` sub-directory. Figure 5 shows the control flow between the target program, kernel, and `cde-exec` during a file-related system call.

For example, when her script requests to load the `/usr/lib/logutils.so` library using an `open` system call, `cde-exec` rewrites the path argument of the `open` call to `cde-package/cde-root/usr/lib/logutils.so` (see Figure 3). This run-time path redirection is essential, because `/usr/lib/logutils.so` probably does not exist on Bob's machine.

2.3 CDE package portability

Alice's CDE package can execute on any Linux machine with an architecture and kernel version that are compatible with its constituent binaries. CDE currently works on 32-bit and 64-bit variants of the x86 architecture (i386 and x86-64, respectively). In general, a 32-bit `cde-exec` can execute 32-bit packaged applications on 32- and 64-bit machines. A 64-bit `cde-exec` can execute both 32-bit and 64-bit packaged applications on a 64-bit machine. Extending CDE to other architectures (e.g., ARM) is straightforward because the `strace` tool that CDE is built upon already works on many architectures. However, CDE packages cannot be transported *across* architectures without using a CPU emulator.

Our portability experiments (§7.1) show that packages are portable across Linux distros released within 5 years of the distro where the package originated. Besides sharing with colleagues like Bob, Alice can also deploy her package to run on a cluster for more computational power or to a public-facing server machine for real-time online monitoring. Since she does not need to install anything as root, she does not risk perturbing existing software on those machines. Also, having her script and all of its dependencies (including the Python interpreter and extension modules) encapsulated within a CDE package makes it somewhat "future-proof" and likely to continue working on her machine even when its version of Python and associated extensions are upgraded in the future.

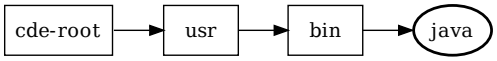

Figure 6: The result of copying a file named `/usr/bin/java` into the `cde-root/` directory.

3 Semi-automated package completion

CDE's primary limitation is that it can only package up files accessed on executed program paths. Thus, programs run from within a CDE package will fail when executing paths that access new files (e.g., libraries, configuration files) that the original execution(s) did not access.

Unfortunately, *no automatic tool* (static or dynamic) can find and package up all the files required to successfully execute all possible program paths, since that problem is undecidable in general. Similarly, it is also impossible to automatically quantify how "complete" a CDE package is or determine what files are missing, since every file-related system call instruction could be invoked with complex or non-deterministic arguments. For example, the Python interpreter executable has only one `dlopen` call site for dynamically loading extension modules, but that `dlopen` could be called many times with different dynamically-generated string arguments derived from script variables or configuration files.

There are two ways to cope with this package incompleteness problem. First, if the user executes additional program paths, then CDE will add new files into the same `cde-package/` directory. However, making repeated executions can get tedious, and it is unclear how many or which paths are necessary to complete the package[2].

Another way to make CDE packages more complete is by manually copying additional files and subdirectories into `cde-package/cde-root/`. For example, while executing a Python script, CDE might automatically copy the few Python standard library files it accesses into, say, `cde-package/cde-root/usr/lib/python/`. To complete the package, the user could copy the entire `/usr/lib/python/` directory into `cde-package/cde-root/` so that *all* Python libraries are present. A user can usually make his/her package complete by copying only a few crucial directories into the package, since programs store all of their files in several top-level directories (see Section 3.3).

However, programs also depend on shared libraries that reside in system-wide directories like `/lib` and `/usr/lib`. Copying all the contents of those directories into a package results in lots of wasted disk space. In Section 3.2, we present an automatic heuristic technique that finds nearly all shared libraries that a program requires and copies them into the package.

[2] similar to trying to achieve 100% coverage during software testing

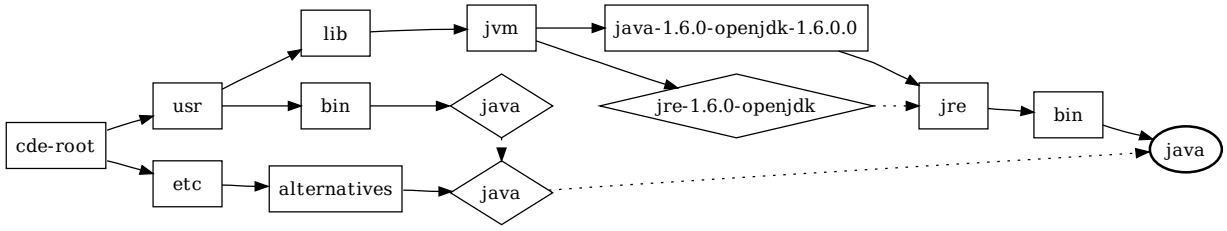

Figure 7: The result of using OKAPI to deep-copy a single `/usr/bin/java` file into `cde-root/`, preserving the exact symlink structure from the original directory tree. Boxes are directories (solid arrows point to their contents), diamonds are symlinks (dashed arrows point to their targets), and the bold ellipse is the actual `java` executable file.

3.1 The OKAPI utility for deep file copying

Before describing our heuristics for completing CDE packages, we first introduce a utility library we built called OKAPI (pronounced "*oh-copy*"), which performs detailed copying of files, directories, and symlinks. OKAPI does one seemingly-simple task that turns out to be tricky in practice: copying a filesystem entity (i.e., a file, directory, or symlink) from one directory to another while fully preserving its original sub-directory and symlink structure (a process that we call *deep-copying*). CDE uses OKAPI to copy files into the `cde-root/` sub-directory when creating a new package, and the support scripts of Sections 3.2 and 3.3 also use OKAPI.

For example, suppose that CDE needs to copy the `/usr/bin/java` executable file into `cde-root/` when it is packaging a Java application. The straightforward way to do this is to use the standard `mkdir` and `cp` utilities. Figure 6 shows the resulting sub-directory structure within `cde-root/`, with the boxes representing directories and the bold ellipse representing the copy of the `java` executable file located at `cde-root/usr/bin/java`. However, it turns out that if CDE were to use this straightforward copying method, the Java application would *fail to run* from within the CDE package! This failure occurs because the `java` executable introspects its own path and uses it as the search path for finding the Java standard libraries. On our Fedora Core 9 machine, the Java standard libraries are actually installed in `/usr/lib/jvm/java-1.6.0-openjdk-1.6.0.0`, so when `java` reads its own path as `/usr/bin/java`, it cannot possibly use that path to find its standard libraries.

In order for Java applications to properly run from within CDE packages, all of their constituent files must be "deep-copied" into the package while replicating their original sub-directory and symlink structures. Figure 7 illustrates the complexity of deep-copying a single file, `/usr/bin/java`, into `cde-root/`. The diamond-shaped nodes represent symlinks, and the dashed arrows point to their targets. Notice how `/usr/bin/java` is a symlink to `/etc/alternatives/java`, which is itself a symlink to `/usr/lib/jvm/jre-1.6.0-openjdk/bin/java`. Another complicating factor is that `/usr/lib/jvm/jre-1.6.0-openjdk` is itself a symlink to the `/usr/lib/jvm/java-1.6.0-openjdk-1.6.0.0/jre/` directory, so the actual `java` executable resides in `/usr/lib/jvm/java-1.6.0-openjdk-1.6.0.0/jre/bin/`. Java can only find its standard libraries when these paths are all faithfully replicated within the CDE package.

The OKAPI utility library automatically performs the deep-copying required to generate the filesystem structure of Figure 7. Its interface is as simple as ordinary `cp`: The caller simply requests for a path to be copied into a target directory, and OKAPI faithfully replicates the sub-directory and symlink structure.

OKAPI performs one additional task: rewriting the contents of symlinks to transform absolute path targets into relative path targets within the destination directory (e.g., `cde-root/`). In our example, `/usr/bin/java` is a symlink to `/etc/alternatives/java`. However, OKAPI cannot simply create the `cde-root/usr/bin/java` symlink to also point to `/etc/alternatives/java`, since that target path is outside of `cde-root/`. Instead, OKAPI must rewrite the symlink target so that it actually refers to `../../etc/alternatives/java`, which is a relative path that points to `cde-root/etc/alternatives/java`.

The details of this particular example are not important, but the high-level message that Figure 7 conveys is that deep-copying even a single file can lead to the creation of over a dozen sub-directories and (possibly-rewritten) symlinks. The problem that OKAPI solves is not Java-specific; we have observed that many real-world Linux applications fail to run from within CDE packages unless their files are deep-copied in this detailed way.

OKAPI is also available as a free standalone command-line tool [1]. To our knowledge, no other Linux file copying tool (e.g., `cp`, `rsync`) can perform the deep-copying and symlink rewriting that OKAPI does.

3.2 Heuristics for copying shared libraries

When Linux starts executing a dynamically-linked executable, the dynamic linker (e.g., `ld-linux*.so*`) finds and loads all shared libraries that are listed in a special `.dynamic` section within the executable file. Running the `ldd` command on the executable shows these start-up library dependencies. When CDE is executing a target program to create a package, CDE finds all of these dependencies as well because they are loaded at start-up time via `open` system calls.

However, programs sometimes load shared libraries in the middle of execution using, say, the `dlopen` function. This run-time loading occurs mostly in GUI programs with a plug-in or extension architecture. For example, when the user instructs Firefox to visit a web page with a Flash animation, Firefox will use `dlopen` to load the Adobe Flash Player shared library. `ldd` will not find that dependency since it is not hard-coded in the `.dynamic` section of the Firefox executable, and CDE will only find that dependency if the user actually visits a Flash-enabled web page while creating a package for Firefox.

We have created a simple heuristic-based script that finds most or all shared libraries that a program requires[3]. The user first creates a base CDE package by executing the target program once (or a few times) and then runs our script, which works as follows:

1. Find all ELF binaries (executables and shared libraries) within the package using the Linux `find` and `file` utilities.
2. For each binary, find all constant strings using the `strings` utility, and look for strings containing "`.so`" since those are likely to be shared libraries.
3. Call the `locate` utility on each candidate shared library string, which returns the *full absolute paths* of all installed shared libraries that match each string.
4. Use OKAPI to copy each library into the package.
5. Repeat this process until no new libraries are found.

This heuristic technique works well in practice because programs often list all of their dependent shared libraries in string *constants* within their binaries. The main exception occurs in dynamic languages like Python or MATLAB, whose programs often dynamically generate shared library paths based on the contents of scripts and configuration files.

Another limitation of this technique is that it is overly conservative and can create larger-than-needed packages, since the `locate` utility can find more libraries than the target program actually needs.

[3] always a superset of the shared libraries that `ldd` finds

3.3 OKAPI-based directory copying script

In general, running an application once under CDE monitoring only packages up a subset of all required files. In our experience, the easiest way to make CDE packages complete is to copy entire sub-directories into the package. To facilitate this process, we created a script that repeatedly calls OKAPI to copy an entire directory at a time into `cde-root/`, automatically following symlinks to other directories and recursively copying as needed.

Although this approach might seem primitive, it is effective in practice because applications often store all of their files in a few top-level directories. When a user inspects the directory structure within `cde-root/`, it is usually obvious where the application's files reside. Thus, the user can run our OKAPI-based script to copy the entirety of those directories into the package.

Evaluation: To demonstrate the efficacy of this approach, we have created complete self-contained CDE packages for six of the largest and most popular Linux applications. For each app, we made an initial packaging run with `cde`, inspected the package contents, and copied at most three directories into the package. The entire packaging process took several minutes of human effort per application. Here are our full results:

- **AbiWord** is a free alternative to Microsoft Word. After an initial packaging run, we saw that some plug-ins were included in the `cde-root/usr/lib/abiword-2.8/plugins` and `cde-root/usr/lib/goffice/0.8.1/plugins` directories. Thus, we copied the entirety of those two original directories into `cde-root/` to complete its package, thereby including all AbiWord plug-ins.
- **Eclipse** is a sophisticated IDE and software development platform. We completed its package by copying the `/usr/lib/eclipse` and `/usr/share/eclipse` directories into `cde-root/`.
- **Firefox** is a popular web browser. We completed its package by copying `/usr/lib/firefox-3.6.18` and `/usr/lib/firefox-addons` into `cde-root/` (plus another directory for the third-party Adobe Flash player plug-in).
- **GIMP** is a sophisticated graphics editing tool. We completed its package by copying `/usr/lib/gimp/2.0` and `/usr/share/gimp/2.0`.
- **Google Earth** is an interactive 3D mapping application. We completed its package by copying `/opt/google/earth` into `cde-root/`.
- **OpenOffice.org** is a free alternative to the Microsoft Office productivity suite. We completed its package by copying the `/usr/lib/openoffice` directory into `cde-root/`.

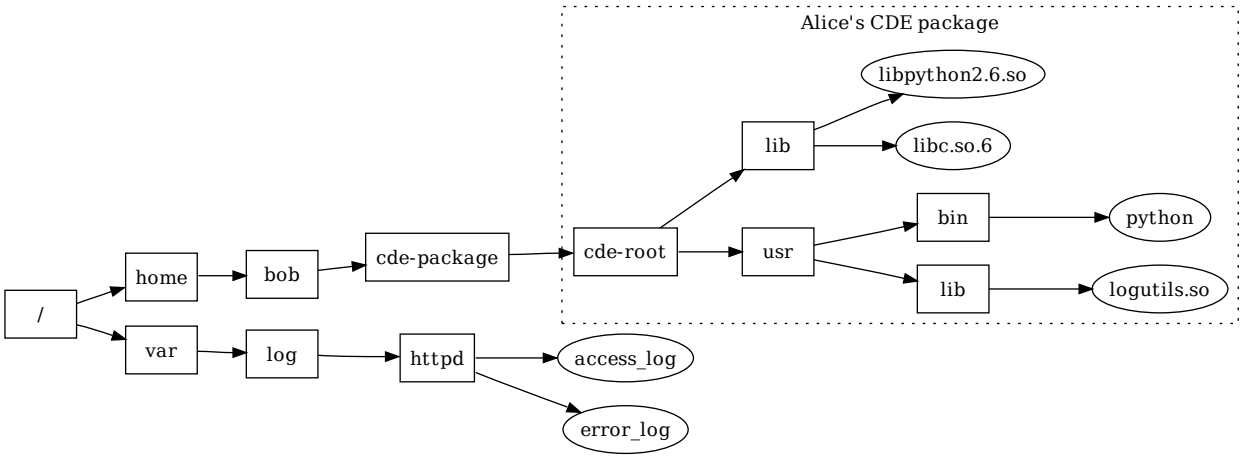

Figure 8: Example filesystem layout on Bob's machine after he receives a CDE package from Alice (boxes are directories, ellipses are files). CDE's seamless execution mode enables Bob to run Alice's packaged script on the log files in `/var/log/httpd/` without first moving those files inside of `cde-root/`.

4 Seamless execution mode

When executing a program from within a package, `cde-exec` redirects all file accesses into the package by default, thereby creating a chroot-like sandbox with `cde-package/cde-root/` as the pseudo-root directory (see Figure 3, Step 3). However, unlike chroot, CDE does not require root access to run, and its sandbox policies are flexible and user-customizable [20].

This default chroot-like execution mode is fine for running self-contained GUI applications like games or web browsers, but it is a somewhat awkward way to run most types of UNIX-style command-line programs that system administrators, developers, and hackers often prefer. If users are running, say, a compiler or command-line image processing utility from within a CDE package, they would need to first move their input data files into the package, run the target program using `cde-exec`, and then move the resulting output data files back out of the package, which is a cumbersome process.

In our Alice-and-Bob example from Section 2 (see Figure 3), if Bob wants to run Alice's anomaly detection script on his own log data (e.g., `bob.log`), he needs to first move his data file inside of `cde-package/cde-root/`, change into the appropriate sub-directory deep within the package, and then run:

```
cde-exec python detect_anomalies.py bob.log
```

In contrast, if Bob had actually installed the proper version of Python and its required extension modules on his machine, then he could run Alice's script from *anywhere* on his filesystem with no restrictions. Some CDE users wanted CDE-packaged programs to behave just like regularly-installed programs rather than requiring input files to be moved inside of a `cde-package/cde-root/` sandbox, so we implemented a new *seamless execution mode* that largely achieves this goal.

Seamless execution mode works using a simple heuristic: If `cde-exec` is being invoked from a directory *not* in the CDE package (i.e., from somewhere else on the user's filesystem), then only redirect a path into `cde-package/cde-root/` if the file that the path refers to actually exists within the package. Otherwise simply leave the path unmodified so that the program can access the file normally. No user intervention is needed in the common case.

The intuition behind why this heuristic works is that when programs request to load libraries and other mandatory components, those files must exist within the package, so their paths are redirected. On the other hand, when programs request to load an input file passed via, say, a command-line argument, that file does not exist within the package, so the original path is used to retrieve it from the native filesystem.

In the example shown in Figure 8, if Bob ran Alice's script to analyze an arbitrary log file on his machine (e.g., his web server log, `/var/log/httpd/access_log`), then `cde-exec` will redirect Python's request for its own libraries (e.g., `/lib/libpython2.6.so` and `/usr/lib/logutils.so`) inside of `cde-root/` since those files exist within the package, but `cde-exec` will *not* redirect `/var/log/httpd/access_log` and instead load the real file from its original location.

Seamless execution mode fails when the user wants the packaged program to access a file from the native filesystem, but an identically-named file actually exists within the package. In the above example, if `cde-package/cde-root/var/`

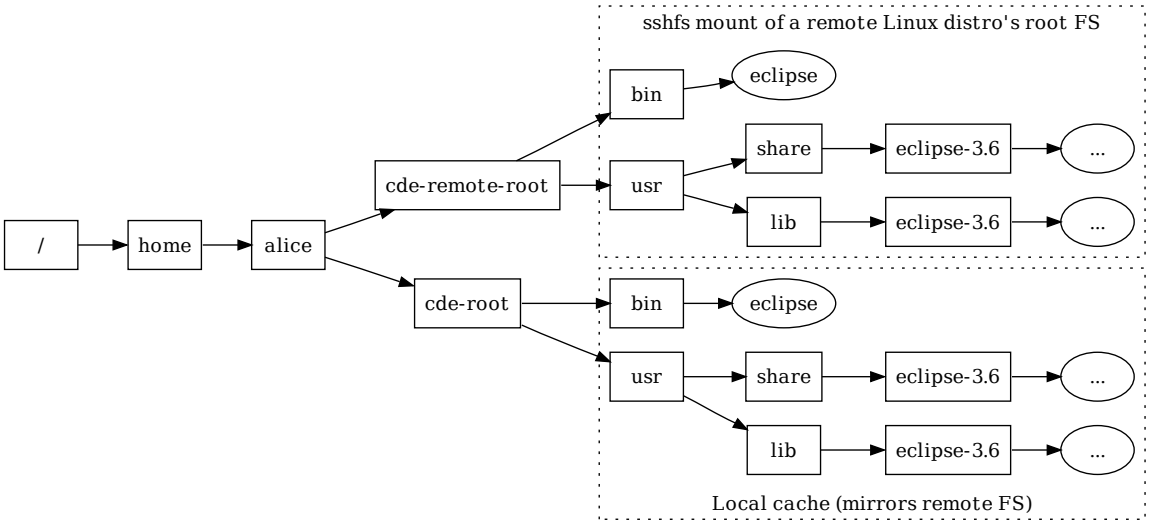

Figure 9: An example use of CDE's streaming mode to run Eclipse 3.6 on any Linux machine without installation. `cde-exec` fetches all dependencies on-demand from a remote Linux distro and stores them in a local cache.

`log/httpd/access_log` existed, then that file would be processed by the Python script instead of `/var/log/httpd/access_log`. There is no automated way to resolve such name conflicts, but `cde-exec` provides a "verbose mode" where it prints out a log of what paths were redirected within the package. The user can inspect that log and then manually write redirection/ignore rules in a configuration file to control which paths `cde-exec` redirects into `cde-root/`. For instance, the user could tell `cde-exec` to *not* redirect any paths starting with `/var/log/httpd/*`.

Using seamless execution mode, our users have been able to run software such as programming language interpreters and compilers, scientific research tools, and sysadmin scripts from CDE packages and have them behave just like regularly-installed programs.

5 On-demand application streaming

We now introduce a new application streaming mode where CDE users can instantly run any Linux application on-demand without having to create, transfer, or install any packages. Figure 2 shows a high-level architectural overview. The basic idea is that a system administrator first installs multiple versions of many popular Linux distros in a "distro farm" in the cloud (or an internal compute cluster). When a user wants to run some application that is available on a particular distro, they use sshfs (an ssh-based network filesystem [9]) to mount the root directory of that distro into a special `cde-remote-root/` mountpoint on their Linux machine. Then the user can use CDE's streaming mode to run any application from that distro locally on their own machine.

5.1 Implementation and example

Figure 9 shows an example of streaming mode. Let's say that Alice wants to run the Eclipse 3.6 IDE on her Linux machine, but the particular distro she is using makes it difficult to obtain all the dependencies required to install Eclipse 3.6. Rather than suffering through dependency hell, Alice can simply connect to a distro in the farm that contains Eclipse 3.6 and then use CDE's streaming mode to "harvest" the required dependencies on-demand.

Alice first mounts the root directory of the remote distro at `cde-remote-root/`. Then she runs "`cde-exec -s eclipse`" (`-s` activates streaming mode). `cde-exec` finds and executes `cde-remote-root/bin/eclipse`. When that executable requests shared libraries, plug-ins, or any other files, `cde-exec` will redirect the respective paths into `cde-remote-root/`, thereby executing the version of Eclipse 3.6 that resides in the cloud distro. However, note that the application is running locally on Alice's machine, not in the cloud.

An astute reader will immediately realize that running applications in this manner can be slow, since files are being accessed from a remote server. While sshfs performs some caching, we have found that it does not work well enough in practice. Thus, we have implemented our own caching layer within CDE: When a remote file is accessed from `cde-remote-root/`, `cde-exec` uses OKAPI to make a deep-copy into a local `cde-root/` directory and then redirects that file's path into `cde-root/`. In streaming mode, `cde-root/` initially starts out empty and then fills up with a subset of files from `cde-remote-root/` that the target program has accessed.

To avoid unnecessary filesystem accesses, CDE's cache also keeps a list of file paths that the target program tried to access from the remote server, even keeping paths for *non-existent files*. On subsequent runs, when the program tries to access one of those paths, `cde-exec` will redirect the path into the local `cde-root/` cache. It is vital to track non-existent files since programs often try to access non-existent files at start-up while doing, say, a search for shared libraries by probing a list of directories in a search path. If CDE did not track non-existent files, then the program would still access the directory entries on the remote server before discovering that those files still do not exist, thus slowing down performance.

With this cache in place, the first time an application is run, all of its dependencies must be downloaded, which could take several seconds to minutes. This one-time delay is unavoidable. However, subsequent runs simply use the files already in the local cache, so they execute at regular `cde-exec` speeds. An added bonus is that even running a *different* application for the first time might still result in some cache hits for, say, generic libraries like `libc`, so the entire application does not need to be downloaded.

Finally, the package incompleteness problem faced by regular CDE (see Section 3) no longer exists in streaming mode. When the target application needs to access new files that do not yet exist in the local cache (e.g., Alice loads a new Eclipse plug-in), those files are transparently fetched from the remote server and cached.

5.2 Synergy with package managers

Nearly all Linux users are currently running one particular distro with one default package manager that they use to install software. For instance, Ubuntu users must use APT, Fedora users must use YUM, SUSE users must use Zypper, Gentoo users must use Portage, etc. Moreover, different releases of the *same* distro contain different software package versions, since distro maintainers add, upgrade, and delete packages in each new release[4].

As long as a piece of software and all of its dependencies are present within the package manager of the exact distro release that a user happens to be using, then installation is trivial. However, as soon as even one dependency cannot be found within the package manager, then users must revert to the arduous task of compiling from source (or configuring a custom package manager).

CDE's streaming mode frees Linux users from this single-distro restriction and allows them to run software that is available within the package manager of any distro in the cloud distro farm. The system administrator is responsible for setting up the farm and provisioning access rights (e.g., ssh keys) to users. Then users can directly install packages in any cloud distro and stream the desired applications to run locally on their own machines.

Philosophically, CDE's streaming mode maximizes user freedom since users are now free to run any application in any package manager from the comfort of their own machines, regardless of which distro they choose to use. CDE complements traditional package managers by leveraging all of the work that the maintainers of each distro have already done and opening up access to users of all other distros. This synergy can potentially eliminate quasi-religious squabbles and flame-wars over the virtues of competing distros or package management systems. Such fighting is unnecessary since CDE allows users to freely choose from amongst all of them.

6 Real-world use cases

Since we released the first version of CDE on November 9, 2010, it has been downloaded at least 3,000 times as of September 2011 [1]. We cannot track how many people have directly checked out its source code from GitHub, though. We have exchanged hundreds of emails with CDE users and discovered six salient real-world use cases as a result of these discussions. Table 1 shows that we used 16 CDE packages, mostly sent in by our users, as benchmarks in the experiments reported in Section 7. They contain software written in diverse programming languages and frameworks. We now summarize the use case categories and benchmarks (highlighted in **bold**).

Distributing research software: The creators of two research tools found CDE online and used it to create portable packages that they uploaded to their websites:

The website for **`graph-tool`**, a Python/C++ module for analyzing graphs, lists these (direct) dependencies: "GCC 4.2 or above, Boost libraries, Python 2.5 or above, expat library, NumPy and SciPy Python modules, GCAL C++ geometry library, and Graphviz with Python bindings enabled." [11] Unsurprisingly, lots of people had trouble compiling it: 47% of all messages on its mailing list (137 out of 289) were questions related to compilation problems. The author of `graph-tool` used CDE to automatically create a portable package (containing 149 shared libraries and 1909 total files) and uploaded it to his website so that users no longer needed to suffer through the pain of manually compiling it.

`arachni`, a Ruby-based tool that audits web application security [10], requires six hard-to-compile Ruby extension modules, some of which depend on versions of Ruby and libraries that are not available in the pack-

[4]We once tried installing a machine learning application that depended on the `libcv` computer vision library. The required `libcv` version was found in the APT repository on Ubuntu 10.04, but it was not found in the repositories on the two immediately neighboring Ubuntu releases: 9.10 and 10.10.

Package name	Description	Dependencies	Creator
Distributing research software			
`arachni`	Web app. security scanner framework [10]	Ruby (+ extensions)	security researcher
`graph-tool`	Lib. for manipulation & analysis of graphs [11]	Python, C++, Boost	math researcher
`pads`	Language for processing ad-hoc data [19]	Perl, ML, Lex, Yacc	self
`saturn`	Static program analysis framework [13]	Perl, ML, Berkeley DB	self
Running production software on incompatible distros			
`meld`	Interactive visual diff and merge tool for text	Python, GTK+	software engineer
`bio-menace`	Classic video game within a MS-DOS emulator	DOSBox, SDL	game enthusiast
`google-earth`	3D interactive map application by Google	shell scripts, OpenGL	self
Creating reproducible computational experiments			
`kpiece`	Robot motion planning algorithm [26]	C++, OpenGL	robotics researcher
`gadm`	Genetic algorithm for social networks [21]	C++, make, R	self
Deploying computations to cluster or cloud			
`ztopo`	Batch processing of topological map images	C++, Qt	graduate student
`klee`	Automatic bug finder & test case generator [16]	C++, LLVM, μClibc	self
Submitting executable bug reports			
`coq-bug-2443`	Incorrect output by Coq proof assistant [2]	ML, Coq	bug reporter
`gcc-bug-46651`	Causes GCC compiler to segfault [3]	gcc	bug reporter
`llvm-bug-8679`	Runs LLVM compiler out of memory [5]	C++, LLVM	bug reporter
Collaborating on class programming projects			
`email-search`	Natural language semantic email search	Python, NLTK, Octave	college student
`vr-osg`	3D virtual reality modeling of home appliances	C++, OpenSceneGraph	college student

Table 1: CDE packages used as benchmarks in our experiments, grouped by use cases. 'self' in the 'Creator' column means package was created by the author; all other packages created by CDE users (mostly people we have never met).

age managers of most modern Linux distributions. Its creator, a security researcher, created and uploaded CDE packages and then sent us a grateful email describing how much effort CDE saved him: *"My guess is that it would take me half the time of the development process to create a self-contained package by hand; which would be an unacceptable and truly scary scenario."*

In addition, we used CDE to create portable binary packages for two of our Stanford colleagues' research tools, which were originally distributed as tarballs of source code: **`pads`** [19] and **`saturn`** [13]. 44% of the messages on the `pads` mailing list (38 / 87) were questions related to troubles with compiling it (22% for `saturn`). Once we successfully compiled these projects (after a few hours of improvising our own hacks since the instructions were outdated), we created CDE packages by running their regression test suites, so that others do not need to suffer through the compilation process.

Even the `saturn` team leader admitted in a public email, *"As it stands the current release likely has problems running on newer systems because of bit rot — some libraries and interfaces have evolved over the past couple of years in ways incompatible with the release."* [7] In contrast, our CDE packages are largely immune to "bit rot" (until the user-kernel ABI changes) because they contain all required dependencies.

Running software on incompatible distros: Even production-quality software might be hard to install on Linux distros with older kernel or library versions, especially when system upgrades are infeasible. For example, an engineer at Cisco wanted to run some new open-source tools on his work machines, but the IT department mandated that those machines run an older, more secure enterprise Linux distro. He could not install the tools on those machines because that older distro did not have up-to-date libraries, and he was not allowed to upgrade. Therefore, he installed a modern distro at home, ran CDE on there to create packages for the tools he wanted to port, and then ran the tools from within the packages on his work machines. He sent us one of the packages, which we used as a benchmark: the **`meld`** visual diff tool.

Hobbyists applied CDE in a similar way: A game enthusiast could only run a classic game (**bio-menace**) within a DOS emulator on one of his Linux machines, so he used CDE to create a package and can now play the game on his other machines. We also helped a user create a portable package for the Google Earth 3D map application (**google-earth**), so he can now run it on older distros whose libraries are incompatible with Google Earth.

Reproducible computational experiments: A fundamental tenet of science is that colleagues should be able to reproduce the results of one's experiments. In the past few years, science journals and CS conferences (e.g., SIGMOD, FSE) have encouraged authors of published papers to put their code and datasets online, so that others can independently re-run, verify, and build upon their experiments. However, it can be hard for people to set up all of the (often-undocumented) dependencies required to re-run experiments. In fact, it can even be difficult to re-run *one's own experiments* in the future, due to inevitable OS and library upgrades. To ensure that he could later re-run and adjust experiments in response to reviewer critiques for a paper submission [16], our groupmate Cristian took the hard drive out of his computer at paper submission time and archived it in his drawer!

In our experience, the results of many computational science experiments can be reproduced within CDE packages since the programs are output-deterministic [15], always producing the same outputs (e.g., statistics, graphs) for a given input. A robotics researcher used CDE to make the experiments for his motion planning paper (**kpiece**) [26] fully-reproducible. Similarly, we helped a social networking researcher create a reproducible package for his genetic algorithm paper (**gadm**) [21].

Deploying computations to cluster or cloud: People working on computational experiments on their desktop machines often want to run them on a cluster for greater performance and parallelism. However, before they can deploy their computations to a cluster or cloud computing (e.g., Amazon EC2), they must first install all of the required executables and dependent libraries on the cluster machines. At best, this process is tedious and time-consuming; at worst, it can be impossible, since regular users often do not have root access on cluster machines.

A user can create a self-contained package using CDE on their desktop machine and then execute that package on the cluster or cloud (possibly many instances in parallel), without needing to install any dependencies or to get root access on the remote machines. For instance, our colleague Peter wanted to use a department-administered 100-CPU cluster to run a parallel image processing job on topological maps (**ztopo**). However, since he did not have root access on those older machines, it was nearly impossible for him to install all of the dependencies required to run his computation, especially the image processing libraries. Peter used CDE to create a package by running his job on a small dataset on his desktop, transferred the package and the complete dataset to the cluster, and then ran 100 instances of it in parallel there.

Similarly, we worked with lab-mates to use CDE to deploy the CPU-intensive **klee** [16] bug finding tool from the desktop to Amazon's EC2 cloud computing service without needing to compile Klee on the cloud machines. Klee can be hard to compile since it depends on LLVM, which is very picky about specific versions of GCC and other build tools being present before it will compile.

Submitting executable bug reports: Bug reporting is a tedious manual process: Users submit reports by writing down the steps for reproduction, exact versions of executables and dependent libraries, (e.g., *"I'm running Java version 1.6.0_13, Eclipse SDK Version 3.6.1, ..."*), and maybe attaching an input file that triggers the bug. Developers often have trouble reproducing bugs based on these hand-written descriptions and end up closing reports as "not reproducible."

CDE offers an easier and more reliable solution: The bug reporter can simply run the command that triggers the bug under CDE supervision to create a CDE package, send that package to the developer, and the developer can re-run that same command on their machine to reproduce the bug. The developer can also modify the input file and command-line parameters and then re-execute, in order to investigate the bug's root cause.

To show that this technique works, we asked people who recently reported bugs to popular open-source projects to use CDE to create executable bug reports. Three volunteers sent us CDE packages, and we were able to reproduce all of their bugs: one that causes the Coq proof assistant to produce incorrect output (**coq-bug-2443**) [2], one that segfaults the GCC compiler (**gcc-bug-46651**) [3], and one that makes the LLVM compiler allocate an enormous amount of memory and crash (**llvm-bug-8679**) [5].

Since CDE is not a record-replay tool, it is not guaranteed to reproduce non-deterministic bugs. However, at least it allows the developer to run the exact versions of the faulting executables and dependent libraries.

Collaborating on class programming projects: Two users sent us CDE packages they created for collaborating on class assignments. Rahul, a Stanford grad student, was using NLTK [22], a Python module for natural language processing, to build a semantic email search engine (**email-search**) for a machine learning class. Despite much struggle, Rahul's two teammates were unable to install NLTK on their Linux machines due to conflicting library versions and dependency hell. This meant that they could only run one instance of the project at a

time on Rahul's laptop for query testing and debugging. When Rahul discovered CDE, he created a package for their project and was able to run it on his two teammates' machines, so that all three of them could test and debug in parallel. Joshua, an undergrad from Mexico, emailed us a similar story about how he used CDE to collaborate on and demo his virtual reality class project (**`vr-osg`**).

7 Evaluation

7.1 Evaluating CDE package portability

To show that CDE packages can successfully execute on a wide range of Linux distros and kernel versions, we tested our benchmark packages on popular distros from the past 5 years. We installed fresh copies of these distros (listed with the versions and release dates of their kernels) on a 3GHz Intel Xeon x86-64 machine:

- Sep 2006 — CentOS 5.5 (Linux 2.6.18)
- Oct 2007 — Fedora Core 8 (Linux 2.6.23)
- Oct 2008 — openSUSE 11.1 (Linux 2.6.27)
- Sep 2009 — Ubuntu 9.10 (Linux 2.6.31)
- Feb 2010 — Mandriva Free Spring (Linux 2.6.33)
- Aug 2010 — Linux Mint 10 (Linux 2.6.35)

We installed 32-bit and 64-bit versions of each distro and executed our 32-bit benchmark packages (those created on 32-bit distros) on the 32-bit versions, and our 64-bit packages on the 64-bit versions. Although all of these distros reside on one physical machine, none of our benchmark packages were created on that machine: CDE users created most of the packages, and we made sure to create our own packages on other machines.

Results: Out of the 96 unique configurations we tested (16 CDE packages each run on 6 distros), all executions succeeded except for one[5]. By "succeeded", we mean that the programs ran correctly, as far as we could observe: Batch programs generated identical outputs across distros; regression tests passed; we could interact normally with the GUI programs; and we could reproduce the symptoms of the executable bug reports.

In addition, we were able to successfully execute all of our 32-bit packages on the 64-bit versions of CentOS, Mandriva, and openSUSE (the other 64-bit distros did not support executing 32-bit binaries).

In sum, we were able to use CDE to successfully execute a diverse set of programs (Table 1) "out-of-the-box" on a variety of Linux distributions from the past 5 years, without performing any installation or configuration.

[5] `vr-osg` failed on Fedora Core 8 with a known error related to graphics drivers.

7.2 Comparing against a one-click installer

To show that the level of portability that CDE enables is substantive, we compare CDE against a representative one-click installer for a commercial application. We installed and ran Google Earth (Version 5.2.1, Sep 2010) on our 6 test distros using the official 32-bit installer from Google. Here is what happened on each distro:

- CentOS (Linux 2.6.18) — installs fine but Google Earth crashes upon start-up with variants of this error message repeated several times, because the GNU Standard C++ Library on this OS is too old:

  ```
  /usr/lib/libstdc++.so.6:
  version 'GLIBCXX_3.4.9' not found
  (required by ./libgoogleearth_free.so)
  ```

- Fedora (Linux 2.6.23) — same error as CentOS
- openSUSE (Linux 2.6.27) — installs and runs fine
- Ubuntu (Linux 2.6.31) — installs and runs fine
- Mandriva (Linux 2.6.33) — installs fine but Google Earth crashes upon start-up with this error message because a required graphics library is missing:

  ```
  error while loading shared libraries:
  libGL.so.1: cannot open shared object
  file: No such file or directory
  ```

- Linux Mint (Linux 2.6.35) — installer program crashes with this cryptic error message because the XML processing library on this OS is *too new* and thus incompatible with the installer:

  ```
  setup.data/setup.xml:1: parser error :
    Document is empty
  setup.data/setup.xml:1: parser error :
    Start tag expected, '<' not found
  Couldn't load 'setup.data/setup.xml'
  ```

In summary, on 4 out of our 6 test distros, a binary installer for the fifth major release of Google Earth (v5.2.1), a popular commercial application developed by a well-known software company, failed in its *sole goal* of allowing the user to run the application, despite advertising that it should work on any Linux 2.6 machine.

If a team of professional Linux developers had this much trouble getting a widely-used commercial application to be portable across distros, then it is unreasonable to expect researchers or hobbyists to be able to easily create portable Linux packages for their prototypes.

In contrast, once we were able to install Google Earth on just *one machine* (Dell desktop running Ubuntu 8.04), we ran it under CDE supervision to create a self-contained package, copied the package to all 6 test distros, and successfully ran Google Earth on all of them without any installation or configuration.

Benchmark	Native run time	CDE slowdown pack	CDE slowdown exec
400.perlbench	23.7s	3.0%	**2.5%**
401.bzip2	47.3s	0.2%	**0.1%**
403.gcc	0.93s	2.7%	**2.2%**
410.bwaves	185.7s	0.2%	**0.3%**
416.gamess	129.9s	0.1%	**0%**
429.mcf	16.2s	2.7%	**0%**
433.milc	15.1s	2%	**0.6%**
434.zeusmp	36.3s	0%	**0%**
435.gromacs	133.9s	0.3%	**0.1%**
436.cactusADM	26.1s	0%	**0%**
437.leslie3d	136.0s	0.1%	**0%**
444.namd	13.9s	3%	**0.3%**
445.gobmk	97.5s	0.4%	**0.2%**
447.dealII	28.7s	0.5%	**0.2%**
450.soplex	5.7s	2.2%	**1.8%**
453.povray	7.8s	2.2%	**1.9%**
454.calculix	1.4s	5%	**4%**
456.hmmer	48.2s	0.2%	**0.1%**
458.sjeng	121.4s	0%	**0.2%**
459.GemsFDTD	55.2s	0.2%	**1.6%**
462.libquantum	1.8s	2%	**0.6%**
464.h264ref	87.2s	0%	**0%**
465.tonto	229.9s	0.8%	**0.4%**
470.lbm	31.9s	0%	**0%**
471.omnetpp	51.0s	0.7%	**0.6%**
473.astar	103.7s	0.2%	**0%**
481.wrf	161.6s	0.2%	**0%**
482.sphinx3	8.8s	3%	**0%**
483.xalancbmk	58.0s	1.2%	**1.8%**

Table 2: Quantifying run-time slowdown of CDE **pack**age creation and **exec**ution within a package on the SPEC CPU2006 benchmarks, using the "train" datasets.

7.3 Evaluating CDE run-time slowdown

The primary drawback of executing a CDE-packaged application is the run-time slowdown due to extra user-kernel context switches. Every time the target application issues a system call, the kernel makes two extra context switches to enter and then exit the `cde-exec` monitoring process, respectively. `cde-exec` performs some computations to calculate path redirections, but its run-time overhead is dominated by context switching[6].

We informally evaluated the run-time slowdown of `cde` and `cde-exec` on 34 diverse Linux applications. In summary, for CPU-bound applications, CDE causes almost no slowdown, but for I/O-bound applications, CDE causes a slowdown of up to ∼30%.

We first ran CDE on the entire SPEC CPU2006 benchmark suite (both integer and floating-point benchmarks) [8]. We chose this suite because it contains CPU-bound applications that are representative of the types of programs that computational scientists and other researchers are likely to run with CDE. For instance, SPEC CPU2006 contains benchmarks for video compression, molecular dynamics simulation, image ray-tracing, combinatorial optimization, and speech recognition.

[6]Disabling path redirection still results in similar overheads.

Command	Native time	CDE slowdown pack	CDE slowdown exec	Syscalls per sec
`gadm` (algorithm)	4187s	0%†	**0%**†	19
`pads` (inferencer)	18.6s	3%†	**1%**†	478
`klee`	7.9s	31%	**2%**†	260
`gadm` (make plots)	7.2s	8%	**2%**†	544
`gadm` (C++ comp)	8.5s	17%	**5%**	1459
`saturn`	222.7s	18%	**18%**	6477
`google-earth`	12.5s	65%	**19%**	7938
`pads` (compiler)	1.7s	59%	**28%**	6969

Table 3: Quantifying run-time slowdown of CDE **pack**age creation and **exec**ution within a package. Each entry reports the mean taken over 5 runs; standard deviations are negligible. Slowdowns marked with † are *not* statistically significant at $p < 0.01$ according to a t-test.

We ran these experiments on a Dell machine with a 2.67GHz Intel Xeon CPU running a 64-bit Ubuntu 10.04 distro (Linux 2.6.32). Each trial was run three times, but the variances in running times were negligible.

Table 2 shows the percentage slowdowns incurred by using `cde` to create each package (the 'pack' column) and by using `cde-exec` to execute each package (the 'exec' column). The 'exec' column slowdowns are shown in **bold** since they are more important for our users: A package is only created once but executed multiple times. In sum, slowdowns ranged from non-existent to ∼4%, which is unsurprising since the SPEC CPU2006 benchmarks were designed to be CPU-bound and not make much use of system calls.

To test more realistic I/O-bound applications, we measured running times for executing the following commands in the five CDE packages that we created (those labeled with "self" in the "Creator" column of Table 1):

- `pads` — Compile a PADS [19] specification into C code (the "`pads` (compiler)" row in Table 3), and then infer a specification from a data file (the "`pads` (inferencer)" row in Table 3).
- `gadm` — Reproduce the GADM experiment [21]: Compile its C++ source code ('C++ comp'), run genetic algorithm ('algorithm'), and use the R statistics software to visualize output data ('make plots').

- `google-earth` — Measure startup time by launching it and then quitting as soon as the initial Earth image finishes rendering and stabilizes.
- `klee` — Use Klee [16] to symbolically execute a C target program (a STUN server) for 100,000 instructions, which generates 21 test cases.
- `saturn` — Run the regression test suite, which contains 69 tests (each is a static program analysis).

We measured the following on a Dell desktop (2GHz Intel x86, 32-bit) running Ubuntu 8.04 (Linux 2.6.24): number of seconds it took to run the original command ('Native time'), percent slowdown vs. native when running a command with `cde` to create a package ('pack'), and percent slowdown when executing the command from within a CDE package with `cde-exec` ('exec'). We ran each benchmark five times under each condition and report mean running times. We used an *independent two-group t-test* [17] to determine whether each slowdown was statistically significant (i.e., whether the means of two sets of runs differed by a non-trivial amount).

Table 3 shows that the more system calls a program issues per second, the more CDE causes it to slow down due to the extra context switches. Creating a CDE package ('pack' column) is slower than executing a program within a package ('exec' column) because CDE must create new sub-directories and copy files into the package.

CDE execution slowdowns ranged from negligible (not statistically significant) to $\sim$30%, depending on system call frequency. As expected, CPU-bound workloads like the `gadm` genetic algorithm and the `pads` inferencer machine learning algorithm had almost no slowdown, while those that were more I/O- and network-intensive (e.g., `google-earth`) had the largest slowdowns.

When using CDE to run GUI applications, we did not notice any loss in interactivity due to the slowdowns. When we navigated around the 3D maps within the `google-earth` GUI, we felt that the CDE-packaged version was just as responsive as the native version. When we ran GUI programs from CDE packages that users sent to us (the `bio-menace` game, `meld` visual diff tool, and `vr-osg`), we also did not perceive any visible lag.

The main caveat of these experiments is that they are informal and meant to characterize "typical-case" behavior rather than being stress tests of worst-case behavior. One could imagine developing adversarial I/O intensive benchmarks that issue tens or hundreds of thousands of system calls per second, which would lead to greater slowdowns. We have not run such experiments yet.

Finally, we also ran some informal performance tests of `cde-exec`'s seamless execution mode. As expected, there were no noticeable differences in running times versus regular `cde-exec`, since the context-switching overhead dominates `cde-exec` computation overhead.

8 Related work

We know of no published system that automatically creates portable software packages *in situ* from a live running machine like CDE does. Existing tools for creating self-contained applications all require the user to manually specify dependencies at package creation time. For example, Mac OS X programmers can create application bundles using Apple's developer tools IDE [6]. Research prototypes like PDS [14], which creates self-contained Windows apps, and the Collective [23], which aggregates a set of software into a portable *virtual appliance*, also require the user to manually specify dependencies.

VMware ThinApp is a commercial tool that automatically creates self-contained portable Windows applications. However, a user can only create a package by having ThinApp monitor the installation of new software [12]. Unlike CDE, ThinApp cannot be used to create packages from existing software already installed on a live machine, which is our most common use case.

Package management systems are often used to install open-source software and their dependencies. Generic package managers exist for all major operating systems (e.g., RPM for Linux, MacPorts for Mac OS X, Cygwin for Windows), and specialized package managers exist for ecosystems surrounding many programming languages (e.g., CPAN for Perl, RubyGems for Ruby) [4].

From the package creator's perspective, it takes time and expertise to manually bundle up one's software and list all dependencies so that it can be integrated into a specific package management system. A banal but tricky detail that package creators must worry about is adhering to platform-specific idioms for pathnames and avoiding hard-coding non-portable paths into their programs [25]. In contrast, creating a CDE package is as easy as running the target program, and hard-coded paths are fine since `cde-exec` redirects all file accesses into the package.

From the user's perspective, package managers work great as long as the *exact* desired versions of software exist within the system. However, version mismatches and conflicts are common frustrations, and installing new software can lead to a library upgrade that breaks existing software [18]. The Nix package manager is a research project that tries to eliminate dependency conflicts via stricter versioning, but it still requires package creators to manually specify dependencies at creation time [18]. In contrast, CDE packages can be run without any installation, configuration, or risk of breaking existing software.

Virtual machine snapshots achieve CDE's main goal of capturing all dependencies required to execute a set of programs on another machine. However, they require the user to always be working within a VM from the start of a project (or else re-install all of their software within a new VM). Also, VM snapshot disk images are (by defi-

nition) larger than the corresponding CDE packages since they must also contain the OS kernel and other extraneous applications. CDE is a more lightweight solution because it enables users to create and run packages natively on their own machines rather than through a VM.

9 Discussion and conclusions

Our design philosophy underlying CDE is that people should be able to package up their Linux software and deploy it to run on other Linux machines with as little effort as possible. However, we are not proposing CDE as a replacement for traditional software installation. CDE packages have a number of limitations. Most notably,

- They are not guaranteed to be complete.
- Their constituent shared libraries are "frozen" and do not receive regular security updates. (Static linking also shares this limitation.)
- They run slower than native applications due to `ptrace` overhead. We measured slowdowns of up to 28% in our informal experiments (§7.3), but slowdowns can be worse for I/O-heavy programs.

Software engineers who are releasing production-quality software should obviously take the time to create and test one-click installers or integrate with package managers. But for the millions of system administrators, research scientists, prototype designers, programming course students and teachers, and hobby hackers who just want to deploy their *ad-hoc* software as quickly as possible, CDE can emulate many of the benefits of traditional software distribution with much less required labor: In just minutes, users can create a base CDE package by running their program under CDE supervision, use our *semi-automated heuristic tools* to make the package complete, deploy to the target Linux machine, and then execute it in *seamless execution mode* to make the target program behave like it was installed normally.

In particular, we believe that the lightweight nature of CDE makes it a useful tool in the Linux system administrator's toolbox. Sysadmins need to rapidly and effectively respond to emergencies, hack together scripts and other utilities on-demand, and run diagnostics without compromising the integrity of production machines. Ad-hoc scripts are notoriously brittle and non-portable across Linux distros due to differences in interpreter versions (e.g., bash vs. dash shell, Python 2.x vs. 3.x), system libraries, and availability of the often-obscure programs that the scripts invoke. Encapsulating scripts and their dependencies within a CDE package can make them portable across distros and minor kernel versions; we have been able to take CDE packages created on 2010-era Linux distros and run them on 2006-era distros [20].

Lessons learned: We would like to conclude by sharing some generalizable system design lessons that we learned throughout the past year of developing CDE.

- First and foremost, start with a conceptually-clear core idea, make it work for basic non-trivial cases, document the still-unimplemented tricky cases, launch your system, and then get feedback from real users. User feedback is by far the easiest way for you to discover what bugs are important to fix and what new features to add next.
- A simple and appealing quick-start webpage guide and screencast video demo are essential for attracting new users. No potential user is going to read through dozens of pages of an academic research paper before deciding to try your system. In short, even hackers need to learn to be great salespeople.
- To maximize your system's usefulness, you must design it to be easy-to-use for beginners but also to allow advanced users to customize it to their liking. One way to accomplish this goal is to have well-designed default settings, which can be adjusted via command-line options or configuration files. The defaults must work well "out-of-the-box" without any tuning, or else beginners will get frustrated.
- Resist the urge to add new features just because they're "interesting", "cool", or "potentially useful". Only add new features when there are compelling real users who demand it. Instead, focus your development efforts on fixing bugs, writing more test cases, improving your documentation, and, most importantly, attracting new users.
- Users are the best sources of bug reports, since they often stress your system in ways that you could have never imagined. Whenever a user reports a bug, try to create a representative minimal test case and add it to your regression test suite.
- If a user has a conceptual misunderstanding of how your system works, then think hard about how you can improve your documentation or default settings to eliminate this misunderstanding.

In sum, get real users, make them happy, and have fun!

Acknowledgments

Special thanks to Dawson Engler for supporting my efforts on this project throughout the past year, to Bill Howe for inspiring me to develop CDE's streaming mode, to Yaroslav Bulatov for being a wonderful CDE power-user and advocate, to Federico D. Sacerdoti (my paper shepherd) for his insightful critiques that greatly improved the prose, and finally to the NSF fellowship for funding this portion of my graduate studies.

References

[1] CDE public source code repository, https://github.com/pgbovine/CDE.

[2] Coq proof assistant: Bug 2443, http://coq.inria.fr/bugs/show_bug.cgi?id=2443.

[3] GCC compiler: Bug 46651, http://gcc.gnu.org/bugzilla/show_bug.cgi?id=46651.

[4] List of software package management systems, http://en.wikipedia.org/wiki/List_of_software_package_management_systems.

[5] LLVM compiler: Bug 8679, http://llvm.org/bugs/show_bug.cgi?id=8679.

[6] Mac OS X Bundle Programming Guide: Introduction, http://developer.apple.com/library/mac/#documentation/CoreFoundation/Conceptual/CFBundles/Introduction/Introduction.html.

[7] Saturn online discussion thread, https://mailman.stanford.edu/pipermail/saturn-discuss/2009-August/000174.html.

[8] Spec cpu2006 benchmarks, http://www.spec.org/cpu2006/.

[9] SSH Filesystem, http://fuse.sourceforge.net/sshfs.html.

[10] arachni project home page, https://github.com/Zapotek/arachni.

[11] graph-tool project home page, http://projects.skewed.de/graph-tool/.

[12] VMware ThinApp User's Guide, http://www.vmware.com/pdf/thinapp46_manual.pdf.

[13] AIKEN, A., BUGRARA, S., DILLIG, I., DILLIG, T., HACKETT, B., AND HAWKINS, P. An overview of the Saturn project. PASTE '07, ACM, pp. 43–48.

[14] ALPERN, B., AUERBACH, J., BALA, V., FRAUENHOFER, T., MUMMERT, T., AND PIGOTT, M. PDS: A virtual execution environment for software deployment. VEE '05, ACM, pp. 175–185.

[15] ALTEKAR, G., AND STOICA, I. ODR: output-deterministic replay for multicore debugging. SOSP '09, ACM, pp. 193–206.

[16] CADAR, C., DUNBAR, D., AND ENGLER, D. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. OSDI '08, USENIX Association, pp. 209–224.

[17] CHAMBERS, J. M. *Statistical Models in S*. CRC Press, Inc., Boca Raton, FL, USA, 1991.

[18] DOLSTRA, E., DE JONGE, M., AND VISSER, E. Nix: A safe and policy-free system for software deployment. In *LISA '04, the 18th USENIX conference on system administration* (2004).

[19] FISHER, K., AND GRUBER, R. PADS: a domain-specific language for processing ad hoc data. PLDI '05, ACM, pp. 295–304.

[20] GUO, P. J., AND ENGLER, D. CDE: Using system call interposition to automatically create portable software packages (short paper). In *USENIX Annual Technical Conference* (June 2011).

[21] LAHIRI, M., AND CEBRIAN, M. The genetic algorithm as a general diffusion model for social networks. In *Proc. of the 24th AAAI Conference on Artificial Intelligence* (2010), AAAI Press.

[22] LOPER, E., AND BIRD, S. NLTK: The Natural Language Toolkit. In *In ACL Workshop on Effective Tools and Methodologies for Teaching NLP and Computational Linguistics* (2002).

[23] SAPUNTZAKIS, C., BRUMLEY, D., CHANDRA, R., ZELDOVICH, N., CHOW, J., LAM, M. S., AND ROSENBLUM, M. Virtual appliances for deploying and maintaining software. In *LISA '03, the 17th USENIX conference on system administration* (2003).

[24] SCAFFIDI, C., SHAW, M., AND MYERS, B. Estimating the numbers of end users and end user programmers. In *IEEE Symposium on Visual Languages and Human-Centric Computing* (2005).

[25] STAELIN, C. mkpkg: A software packaging tool. In *LISA '98, the 12th USENIX conference on system administration* (1998).

[26] SUCAN, I. A., AND KAVRAKI, L. E. Kinodynamic motion planning by interior-exterior cell exploration. In *Int'l Workshop on the Algorithmic Foundations of Robotics* (2008), pp. 449–464.