Safe and Scalable Build Automation through Virtualization

Martin v. Löwis, Peter Tröger

martin.vonloewis@hpi.uni-potsdam.de
peter.troeger@hpi.uni-potsdam.de

Abstract: Hardware virtualization is often used to achieve isolation of one application from another, while still allowing to run the applications on the same physical hardware for better utilization; using separate machines might have been an alternative. Often, people associate this scenario with long-running tasks, such as web servers or other services offered in the network. In this paper, we discuss a use case of virtualization that is quite different, where virtual machines are created and destroyed so quickly that using physical hardware would have been feasible at all. The specific application that we consider is build automation in the Python Package Index, as implemented in the HPI Future SOC Lab.

1 Introduction

For several decades, people have been using hardware virtualization as a concept for isolation and multiprogramming [ABCC66, Gol74, RG05]. As mainstream architectures now allow for efficient virtualization [AA06], hypervisors are today a standard approach for server consolidation. Example technologies for x86 environments are VMWare, VirtualBox, Hyper-V, Xen, or KVM.

In this paper, we want to discuss a practical use case for virtualization that involves balancing between security and correctness on the one hand, and performance on the other hand. We show how virtualization as isolation approach can help to provide a stable automated software build environment, and suggest potential areas for improvement in the hypervisor implementations for this class of applications.

The following section 2 presents the use case for virtualization in the PyPI build automation environment. Section 3 then explains the resulting performance impact of the virtualization environment itself, and presents some initial solution approaches. Section 4 discusses some related research work.

All experiments for realizing build automation based on virtualization were performed in the HPI Future SOC Lab environment [MPZ+11].

erschienen im Tagungsband der INFORMATIK 2011
Lecture Notes in Informatics, Band P192
ISBN 978-3-88579-286-4

2 PyPI Build Automation

Our work for an automated build system is intended for the Python Package Index[1] (PyPI, formerly known as the Cheeseshop) a central repository for software libraries and applications. Currently, it hosts about 14,000 packages, maintained by ca. 5,000 registered developers. PyPI has a steady grow in the number of software releases, which includes both initial and update releases (see Figure 1). While the packages are written primarily in Python, other languages (in particular C/C++ and JavaScript) are also used.

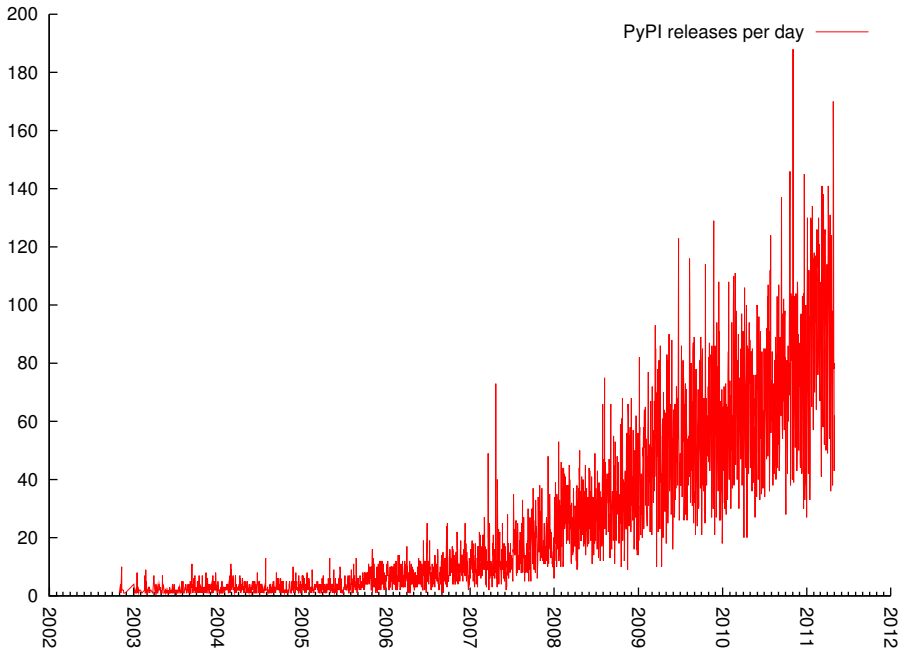

Figure 1: Release Statistics

In the Python ecosystem, the installation procedure for a package is typically written in Python itself (see Table 2). The installation steps range from copying files around on disk, over generating files on-the-fly during installation, to invoking the C compiler to build machine code for the target system. Part of the installation procedure is also to make sure that all dependencies of the package are satisfied, i.e. that appropriate versions of prerequisite packages are installed. The list of specific dependencies required by a package may itself be computed only at installation time, e.g. when a dependency is conditional on the kind of operating system or the specific version of the Python interpreter.

Linux distributions solve the installation issues with a slightly different approach: instead of installing source code, users install pre-compiled binary packages. In these binary pack-

[1] `http://pypi.python.org/`

Library	Documentation
distutils	`http://docs.python.org/distutils/`
setuptools	`http://pypi.python.org/pypi/setuptools/`
distribute	`http://pypi.python.org/pypi/distribute/`
zc.buildout	`http://pypi.python.org/pypi/zc.buildout/`
stdeb	`https://github.com/astraw/stdeb/`

Figure 2: Python Package Installation Support

ages, most build steps have already been performed, resulting in a package that is specific for the Linux distribution. As a consequence, it becomes possible to declare dependencies in a strictly static manner. For example, the Debian family of operating systems uses the .deb file format [JS11] for packaging. Debian users often prefer to install "native" Debian packages, instead of having to run custom installation procedures. Specific packages need to be built for every Debian release, as well as for every distribution derived from Debian, such as the various Ubuntu releases.

The objective of the intended build system is to provide all PyPI packages as Debian packages, enabling users to use standard installation procedures. In building these packages, various issues have to be considered:

- Debian distributions already provide Debian packages for some of the PyPI packages. Only a small fractions is supported, though, and often, the Debian release includes an older version than is available on PyPI. Where an officially-maintained package is already provided by Debian, this project should not strive to provide a competing binary package.
- Debian is available for multiple CPU architectures, most notably the Intel32 architecture and the AMD64 architecture. Where the Python package uses C code, the resulting binary packages become architecture-dependent.
- Each Debian release upgrades many of its libraries, resulting potentially in incompatibilities of applications across Debian releases. Binary packages built for Debian must therefore be built for a specific Debian release, to guarantee that the right set of base libraries (in particular, header files and shared libraries) is used during the build.

As a consequence, the project needs to build the whole set of PyPI packages for every new Debian and Ubuntu release. In addition, each package needs to be built for all target systems whenever a new release of the package is made. The build infrastructure should be fully automated, so that Debian packages become available quickly after a new release. This may result in packages that have a lower quality than the manually-maintained Debian packages. For example, the manually-created package might provide the documentation for the package as a separate binary package, whereas the automatically created one fails to package the documentation at all, as the standard installation procedure does not cope

ISBN 978-3-88579-286-4

with documentation building. In exchange, users will have to wait (often for many months) for a new release of the Debian package.

2.1 Issues with automated PyPI installations

In order to run a fully-automated installation procedure, a number of issues need to be considered. First, the Python packages often do not contain enough information to build a "correct" Debian package. The *stdeb* package (see Table 2) copes for this lack of information, by extracting as much meta-data as feasible from the Python package, and filling the rest with dummy data.

Second, the build procedure may run untrusted code. PyPI is open for anybody to submit code; the manual review process to establish trust that the Debian maintainers perform would not be used when creating packages automatically. Therefore, a malicious user may introduce malware in the build process. Even if not assuming malicious users, installation procedures often leave the system in a modified state that cannot be easily reverted: files may be placed in central library folders, and libraries be overwritten; services may be started; user accounts created; etc.

Third, packages may fail to properly declare their dependencies. This should result in failing tests, but will so only if the build machine doesn't have the dependencies installed. In order to detect this issue, each build should start with a minimum set of installed packages.

To resolve the second and third issue, a clean file system image is needed for every new build, on a machine that does not have sensitive data on it, and the system performing the build needs to be decoupled from the network in a way to prevent malicious code to perform an attack on the local network. The only data surviving each build should be the log files of the build, and, if the build was successful, the resulting binary package.

With the need for a file system reset comes the fourth issue of build times. Many of the build procedures complete within a few seconds, as shown in Figure 3. The time to reset the file system to a clean state should not be significantly larger than the actual build times. More specifically, the packages released on a single day should become available in binary form on the same day (or else the build process will never catch up), and the complete rebuild necessary for a new operating system release should complete well before the next release of the operating system.

3 Virtualization As Solution

To implement this project, we have decided to use virtual machines as an approach. Virtual machines have been demonstrated to provide the necessary isolation for the execution of untrusted or malicious code [SC09].

With respect to the issues discussed above, we have considered a number of design alternatives that we would like to present.

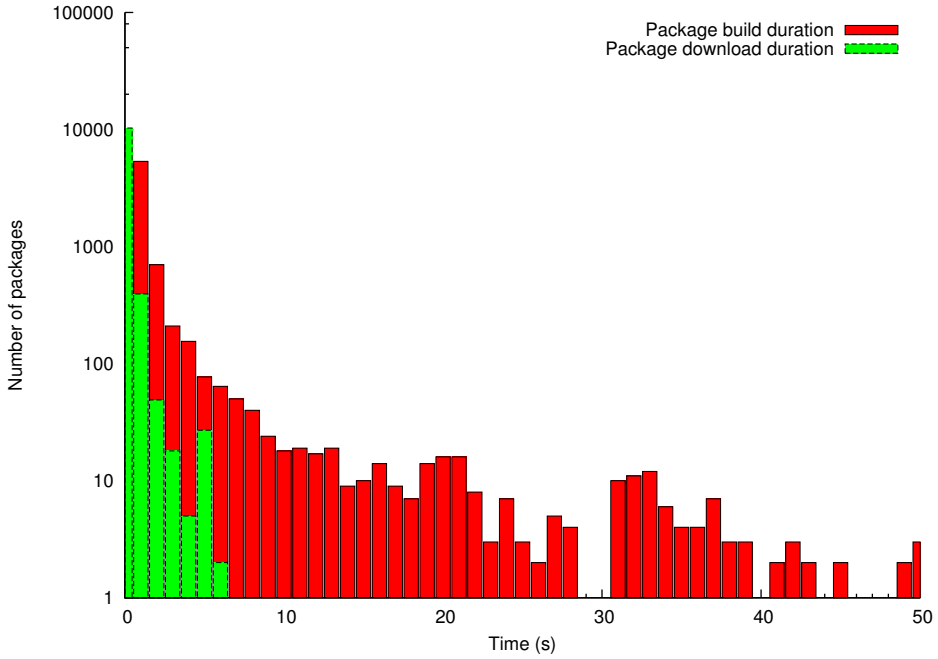

Figure 3: Build and Download Time Histogram

The main challenge of the project is to start each build process from a clean, minimal system. Each declared pre-requisite package ought to be installed during the build process, followed by a download of the package to be built, followed by the actual build steps, followed by an upload of the build results.

In order to reset the file system, we identified three possible strategies:

- As Linux supports overlay mounts, a large-enough ram-disk can be laid over the boot disk. All installation and build steps will write to the ram disk, which is discarded after the build. This does not use special hypervisor functionality, but may run into semantic limitations of overlay mounts.
- Hard disk images of the guest system are often represented as files in the host system. Before booting the virtual machine, a copy of the hard disk could be made, and be restored after the build is complete. This is likely an expensive operation.
- Some hypervisor implementations support disk snapshot mechanisms, storing modifications to the base disk image in a separate file. Rolling back the disk state is as simple as deleting the delta information. The cost of this approach is in the overhead of virtually merging the base image with the delta modifications, which is necessary for each read access.

As a specific detail of our project, consider that we are using host systems with very large amounts of physical memory. Therefore, it becomes feasible to store all virtual hard disk information of the guest systems in physical RAM of the host machine. As we would also want to run multiple build processes in parallel, it would be best if the base image could be shared in a read-only manner in RAM across all guest systems. The modifications then can still go into RAM as well, subject to the limitation of the physical RAM. Our current hardware has 48 logical processors in system of 256 GiB of RAM, allowing for, e.g. 40 virtual machines running with 5 GiB of RAM each.

Another issue is the communication with the guest system: as systems start in a clean state, they will have no indication what the build process is that they need to run. We have identified two approaches:

- Upon start-up, each node contacts a service at a well-known IP address, to download configuration information, prerequisite packages, the source package, and to upload results to. As the network of the guest system needs to be isolated from the internet, setting up the necessary infrastructure may be challenging.
- Some hypervisors support mounting parts of the host file system into the guest system, often using special paravirtualized file system drivers. It would be possible to provide the guest system with a read-only volume containing all the files it may need (e.g. providing a complete mirror of the respective Debian release for use in */etc/apt/sources.list*), as well as a separate volume where output files are stored (initially empty).

The guest system gets pre-configured (in its clean state) to perform a build script during start-up which actually triggers the build process. When the build process is complete, the build output gets uploaded, and the machine shuts down. As a further optimization, we are considering to have the clean image already represented a booted machine state (provided sufficient support in the hypervisor).

We are currently using Oracle VirtualBox[2] as the hypervisor technology, as it supports delta storage for disk snapshots, as well as memory snapshots, and is scriptable using a command line interface and a Python API.

4 Related Work

Software build automation is identified to be one of the relevant research aspect for the software engineering community [Sha01]. Beside classical approaches as with *automake* and *autoconf*, there are more elaborated systems targeting different programming languages [MO05].

Continuous integration as another software engineering aspect focuses on the utilization of build automation for continuous quality control [KPYL08]. With a standardized support

[2] `http://www.virtualbox.org/`

ISBN 978-3-88579-286-4

for test suite execution in PyPI-hosted packages, virtualization again could help in isolating the test runs from each other.

The specific aspect of fast machine startups, as discussed in Section 3, is well-researched in the context of preventing software-aging effects in long-running systems. The *software rejuvenation* approaches, e.g. [VHHT01], relies on direct modification of the operating system start-up procedures or on snapshot mechanisms.

5 Conclusion

We have demonstrated a use case of virtualization that requires creation and deletion of virtual machines in quick succession. The motivation for using virtual machines is the execution of untrusted code in a completely automated fashion; the specific example is that of building Debian packages from the Python Package Index.

We have discussed alternatives in applying hypervisor features for this application. We believe that increase in available RAM will change the operation models in data centers soon, in a way similar to how the advent of many-core systems affected the world of programming models.

In continuing this project, we will need to gain operational experience with such an infrastructure, over a period of several months.

References

[AA06] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. *SIGOPS Operating Systems Review*, 40:2–13, October 2006.

[ABCC66] R. Adair, R. Bayles, L. Comeau, and R. Creasy. A Virtual Machine System for the 360/40. Technical Report G320-2007, IBM, 1966.

[Gol74] Robert Goldberg. Survey of Virtual Machine Research. *IEEE Computer*, 7:34–45, June 1974.

[JS11] Ian Jackson and Christian Schwarz. *Debian Policy Manual v3.9.2.0*, 4 2011.

[KPYL08] Seojin Kim, Sungjin Park, Jeonghyun Yun, and Younghoo Lee. Automated Continuous Integration of Component-Based Software: An Industrial Experience. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 423–426, Washington, DC, USA, 2008. IEEE Computer Society.

[MO05] Vincent Massol and Timothy O'Brien. *Maven: A Developer's Notebook (Developer's Notebooks)*. O'Reilly Media, Inc., 2005.

[MPZ+11] Christoph Meinel, Andreas Polze, Alexander Zeier, Gerhard Oswald, Dieter Herzog, Volker Smid, Doc D Errico, and Zahid Hussain. Proceedings of the Fall 2010 Future SOC Lab Day. Technical Report 42, Hasso-Plattner-Institute, 2011.

[RG05] M. Rosenblum and T. Garfinkel. Virtual machine monitors: current technology and future trends. *Computer*, 38:39–47, May 2005.

ISBN 978-3-88579-286-4

[SC09] Gaurav Somani and Sanjay Chaudhary. Application Performance Isolation in Virtualization. In *2009 IEEE International Conference on Cloud Computing*, pages 41–48, Washington, DC, USA, 2009. IEEE Computer Society.

[Sha01] Mary Shaw. The coming-of-age of software architecture research. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 656–, Washington, DC, USA, 2001. IEEE Computer Society.

[VHHT01] Kalyanaraman Vaidyanathan, Richard E. Harper, Steven W. Hunter, and Kishor S. Trivedi. Analysis and implementation of software rejuvenation in cluster systems. In *Proceedings of the 2001 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '01, pages 62–71, New York, NY, USA, 2001. ACM.

ISBN 978-3-88579-286-4