μCernVM: Slashing the Cost of Building and Deploying Virtual Machines

J Blomer, D Berzano, P Buncic, I Charalampidis, G Ganis, G Lestaris, R Meusel, V Nicolaou

jblomer@cern.ch

CERN PH-SFT
CHEP 2013, Amsterdam
The Cern Virtual Machine

CernVM 2.X

- User Data (EC2, Openstack)
- Contextualization
- System Libraries & Tools
- Operating System Kernel
- CernVM-FS
- Fuse
- HTTP Cache Hierarchy

- Used on laptops, clouds, volunteers’ computers (LHC@Home 2.0)
- Uniform and portable environment for physics data processing
- “Just enough operating system” derived from application dependencies
CernVM 2.X series

- **Versatile image**
  supports all hypervisors
  supports all LHC experiments
  various roles: desktop, batch, ... 

- **Strongly versioned** components
  Single version number defines VM
  (by Conary package manager)

- **Scientific Linux 5 based**

- **Significant effort to**
  build, test, deploy images

Goals for CernVM 3

1. Update base platform
   Scientific Linux 6, RPM
   support for cloud-init

2. Instant feedback during
   appliance development:
   minutes instead of hours

3. Instant instantiation
   of virtual machines:
   seconds instead of minutes

Idea: CernVM-FS can provide operating system on demand
Building blocks of CernVM 3

Twofold system: $\mu$CernVM boot loader + OS delivered by CernVM-FS
Building blocks of CernVM 3

Twofold system: $\mu$CernVM boot loader + OS delivered by CernVM-FS

⇒ Drastic reduction in size

From “just enough operating system” to “operating system on demand”

400 MB image (compressed) $\leftrightarrow$ 12 MB image + 100 MB cache
Part I: $\mu$CernVM Boot Loader

Part II: Operating System on CernVM-FS
CernVM Kernel: 3.10 long-term kernel (2 year support)
Features: KSM, zRam, THP, cgroups, X32-ABI
Extra modules: AUFS, VMware drivers, VBox drivers, OpenAFS

“Virtualization-friendly”, minimal set of device drivers:
100 modules / 8 MB as compared to 2000 modules / 120 MB in SL6

1. Execute SYSLINUX boot loader
2. Decompress and load Linux kernel
3. Decompress init ramdisk, execute customized /init
   a) Start networking
   b) Contextualize (supports EC2, OpenStack, OpenNebula, vSphere)
   c) [Partition and] [format and] mount scratch space
   d) Mount CernVM-FS
   e) Mount AUFS root file system stack
   f) Change root file system and start operating system
Booting $\mu$CernVM

* Welcome to micro-CernVM
* Beta release 1.14-1.cernvm.x86_64

[INF] Loading predefined modules... check
[INF] Starting networking... check
[INF] Getting time from ptbtime1.ptb.de... check
[INF] Contextualizing VM... (none)
[INF] Partitioning /dev/sda... check
[INF] Formatting /dev/sda1... check
[INF] Mounting root filesystem... check
[INF] Starting CernVM File System... connected to cernvm-devel.cern.ch
[INF] Pinning core file set... check
[INF] Posting kernel modules... check
[INF] Booting CERN Virtual Machine 3.0.0.0

mount: mount point /proc/bus/usb does not exist
       Welcome to Scientific Linux

Starting udev: _
μCernVM Root File System Stack

CernVM-FS features targeted to loading the OS:

- Closing all read-write file descriptors in order to unravel file system stack on shutdown
- Redirect syslog messages
- In-flight change of DNS server
- GID / UID mappings

⇒ This file system stack requires special support from a read-only Fuse branch since it is started before the operating system.
Part I: $\mu$CernVM Boot Loader

Part II: Operating System on CernVM-FS
Scientific Linux on CernVM-FS

General idea: Install packages with yum into a CernVM-FS chroot jail

Problem: Typical package repositories are not designed to preserve an environment

The CernVM 3 build process **ensures strong versioning** on three levels

1. cernvm-system meta RPM
   fully versioned dependency closure

2. Named branches in the CernVM-FS repository

3. Versioned snapshots provided by CernVM-FS
   allow the very same image to instantiate any cernvm-system version
   helpful for **long-term data preservation**

---

production

3.0.64.0

testing

3.1.32.0

development

3.2.1.0

cernvm-system
Scientific Linux on CernVM-FS

General idea: Install packages with yum into a CernVM-FS chroot jail

Problem: Typical package repositories are not designed to preserve an environment

The CernVM 3 build process ensures strong versioning on three levels

1. cernvm-system meta RPM
   fully versioned dependency closure

2. Named branches in the CernVM-FS repository

3. Versioned snapshots provided by CernVM-FS
   allow the very same image to instantiate any cernvm-system version
   helpful for long-term data preservation
Build Process: Scientific Linux on CernVM-FS

Maintenance of the repository **should not** become a Linux distributor’s job

**But:** should be reproducible and well-documented

**Idea:** automatically generate a **fully versioned, closed** package list from a “shopping list” of unversioned packages

Scientific Linux  EPEL  CernVM Extras ($\approx 50$)

Formulate dependencies as Integer Linear Program

yum install on CernVM-FS

gcc  emacs  ...

Dependency Closure

Package Archive
Virtual Machine Life Cycle

A virtual machine passes through various different stages throughout its life. These stages are just a logical separation of the fundamental procedures that are common for the maintenance of every virtual machine (VM). They are usually independent and are associated with a specific set of tools. For instance, the life of the VM begins when the specifications of the build process are prepared and stored in a reference database, and it is terminated after it has completed the job it was instantiated for. In order to find an optimal solution it is important to identify those stages, the tools associated with them and their dependencies. This way the appropriate tools can be grouped with the stages and form a stand-alone and independently-maintained component.

In the CernVM Project we pass through all stages of the every time we release a new version. In our case, after a VM instance completes its cycle, user feedback is processed and a new development cycle begins. Because of this cycling pattern, we decided to use the term lifecycle to refer to the life of CernVM. This lifecycle can be split into two logical sub-cycles: the development cycle and the deployment cycle (Figure 1).

The development cycle begins with the definition of the specifications and finishes with the production of the distributable VM media. This cycle is performed entirely inside the CernVM infrastructure.

The deployment cycle begins with the instantiation of the released image and finishes with the termination of the instance. This cycle is performed outside the CernVM infrastructure, such as a public or private cloud infrastructure (e.g. Amazon or OpenNebula) or an individual computer (e.g. desktop hypervisors or a small computer farm). In all these cases, the OS needs to contact the CernVM infrastructure in order to obtain contextualization information and software packages from our repository.

The two cycles are connected via two intermediate stages: the release of the produced image to the public and the feedback that is collected from the users and triggers a new development cycle. The two stages are in the borders that split the private infrastructure and the public.

As was mentioned before, each stage is independent and is typically supported by a number of specialized tools.

Plan: This is a stage on which the desired functionality of the VM is planned. The resulting
The Virtual Machine Lifecycle

A virtual machine passes through various different stages throughout its life. These stages are just a logical separation of the fundamental procedures that are common for the maintenance of every virtual machine (VM). They are usually independent and are associated with a specific set of tools. For instance, the life of the VM begins when the specifications of the build process are prepared and stored in a reference database, and it is terminated after it has completed the job it was instantiated for. In order to find an optimal solution it is important to identify those stages, the tools associated with them and their dependencies. This way the appropriate tools can be grouped with the stages and form a stand-alone and independently-maintained component.

In the CernVM Project we pass through all stages of the every time we release a new version. In our case, after a VM instance completes its cycle, user feedback is processed and a new development cycle begins. Because of this cycling pattern, we decided to use the term lifecycle to refer to the life of CernVM. This lifecycle can be split into two logical sub-cycles: the development cycle and the deployment cycle (Figure 1).

The development cycle begins with the definition of the specifications and finishes with the production of the distributable VM media. This cycle is performed entirely inside the CernVM infrastructure.

The deployment cycle begins with the instantiation of the released image and finishes with the termination of the instance. This cycle is performed outside the CernVM infrastructure, such as a public or private cloud infrastructure (e.g. Amazon or OpenNebula) or an individual computer (e.g. desktop hypervisors or a small computer farm). In all these cases, the OS needs to contact the CernVM infrastructure in order to obtain contextualization information and software packages from our repository.

The two cycles are connected via two intermediate stages: The release of the produced image to the public and the feedback that is collected from the users and triggers a new development cycle. The two stages are in the borders that split the private infrastructure and the public. As was mentioned before, each stage is independent and is typically supported by a number of specialized tools.

Plan:

This is a stage on which the desired functionality of the VM is planned. The resulting

Avoids: Image Building
Solves: Image Distribution
Options for updating: stay, diverge, rebase

Rebase high-level perspective:
1. On first boot, CernVM selects and pins newest available version
2. Automatic update notifications
3. Applying updates requires a reboot
   Most security critical updates require a reboot anyway

µCernVM Bootloader
- boot partition read-only, updates dropped on ephemeral storage
- 2 phase boot:
  start old kernel and ramdisk, kexec into updated version

CernVM-FS OS Repository
- Mount updated CernVM-FS snapshot
- Conflict resolution wrt.
  local changes
  1. keep local configuration
  2. map user/group ids
  3. merge rpm database
Options for updating: stay, diverge, rebase

Rebase high-level perspective:
1. On first boot, CernVM selects and pins newest available version
2. Automatic update notifications
3. Applying updates requires a reboot
   Most security critical updates require a reboot anyway

\(\mu\text{CernVM Bootloader}\)
- boot partition read-only, updates dropped on ephemeral storage
- 2 phase boot:
  start old kernel and ramdisk, \texttt{kexec} into updated version

\(\text{CernVM-FS OS Repository}\)
- Mount updated CernVM-FS snapshot
- Conflict resolution wrt. local changes
  1. keep local configuration
  2. map user/group ids
  3. merge rpm database
Instantiating a Virtual Machine in CernVM Online

Web browser plugin developed by Ioannis Charalampidis for
http://crowdcrafting.org/app/cernvm
Example

Instantiating a Virtual Machine in CernVM Online

Welcome to uCernVM

Press [Tab] to edit options

Loading

Automatic boot in 1 second...
Example

Instantiating a Virtual Machine in CernVM Online
1. CernVM 3 releases are created in few minutes
2. A single virtual machine image of only 12 MB can instantiate any CernVM 3 version ever released

The reality check
- Build CernVM 3 components
- PROOF and HTCondor clusters
- ATLAS, ALICE event viewers
- CMS, LHCb reconstruction

Next steps
- Systematic testing!
- Use WLCG infrastructure for CernVM-FS OS repository

We are happy for feedback!
Please find beta releases and build system sources under:
http://cernvm.cern.ch/portal/ucernvm
https://github.com/cernvm
More about **CernVM Online**
G Lestaris et al.: *CernVM Online and Cloud Gateway: a uniform interface for CernVM contextualization and deployment*
http://chep2013.org/contrib/185

More about **PROOF as a Service**
D Berzano et al.: *PROOF as a Service on the Cloud: a Virtual Analysis Facility based on the CernVM ecosystem*
http://chep2013.org/contrib/185
3 Backup Slides
<table>
<thead>
<tr>
<th>Hypervisor / Cloud Controller</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>VirtualBox</td>
<td>✓</td>
</tr>
<tr>
<td>VMware</td>
<td>✓</td>
</tr>
<tr>
<td>KVM</td>
<td>✓</td>
</tr>
<tr>
<td>Xen</td>
<td>✓</td>
</tr>
<tr>
<td>Microsoft HyperV</td>
<td>?</td>
</tr>
<tr>
<td>Parallels</td>
<td>⬤ 4</td>
</tr>
<tr>
<td>Openstack</td>
<td>✓</td>
</tr>
<tr>
<td>OpenNebula</td>
<td>✓ 3</td>
</tr>
<tr>
<td>Amazon EC2</td>
<td>✓ 1</td>
</tr>
<tr>
<td>Google Compute Engine</td>
<td>⬤ 2</td>
</tr>
</tbody>
</table>

1. Only tested with ephemeral storage, not with EBS backed instances
2. Waiting for custom kernel support
3. Only amiconfig contextualization
4. Unclear license of the guest additions
Normalized (Integer) Linear Program:

\[
\begin{align*}
\text{Minimize } & \quad (c_1 \cdots c_n) \cdot \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \\
\text{subject to } & \quad \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \leq \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix}
\end{align*}
\]

Here: every available (package, version) is mapped to a \( x_i \in \{0, 1\} \).

Cost vector: newer versions are cheaper than older versions.
(Obviously: less packages cheaper than more packages.)

Dependencies:
- Package \( x_a \) requires \( x_b \) or \( x_c \): \( x_b + x_c - x_a \geq 0 \).
- Packages \( x_a \) and \( x_b \) conflict: \( x_a + x_b \leq 1 \).

Figures

\( \approx 17000 \) available packages (\( n = 17000 \)), \( 500 \) packages on “shopping list”
\( \approx 160000 \) inequalities (\( m = 160000 \)), solving time <10 s (glpk)
Meta RPM: \( \approx 1000 \) fully versioned packages, dependency closure