Bootstrapping Debian for a new architecture

Pietro Abate

Universite Paris Diderot / Irill

24-11-2012
Acknowledgements

- Most of the work done by Johannes Schauer during its GSOC 2012
- Mentoring myself and Wookey
Problem

- Debian was ported to more than 20 architectures so the process is executed roughly once per year.
- Debian packages are neither made to be cross compilable nor to be built without an existing full Debian installation.
- For each new port a set of source packages has to be cross compiled and/or built manually.
- Bootstrap a new architecture often involves foreign distributions and a lot of hacking.
Wish List

- Porting Debian to a new architecture should be less time consuming and less problematic.
- No foreign distributions during porting (self hosted).
- Automatic cross compiling for architectures that cannot build themselves.
- Sub-arch builds optimized for a specific CPU should be easier.
The final Goal: Deducing a build order

1. Step zero: Bare metal.
2. Cross compilation: create a minimal build system (XC).
3. Automatic (cross) compile XC.
4. Switch to native compilation.
5. Find the largest number of sources that can be natively built (NC).
6. Automatic compile NC (we need a build order).

We need correct Multi-Arch annotations and build profiles.
Stage Compilation

- Directly building fully fledged binary packages is impossible because of the presence of build dependency cycles.
- We need to weak build dependencies in order to remove these dependency cycles.
- Build Profiles are the proposed solution.
  - A build profile is a global build dependency filter.
  - It is the form: `Build-Depends: foo [i386 arm] <!stage1>`
  - The format similar to architecture specifiers.
Stage Compilation

- Nothingness
- Build more packages
- Stage 1 compilation
- Reduced Binary Set
- Normal compilation
- Cross or native compile
- Reduced set of functionalities
- Full Binary Set
Why we need Cross Compilation?

- A new architecture cannot be bootstrapped from thin air
- At least a minimal system must be cross built
- This system should be large enough to compile the entire distribution
- Native compilation should be preferred over cross compilation
Cross compilation. Package selection.

The minimal set of packages that must be cross compiled ($XC$) are those with the following properties:

- **Essential:** yes
- **Build-Essential:** yes
- **Priority:** required

Plus debhelper as 79% of the archive depend on it.
Minimal build system

How many packages are in the minimal build system for Sid?

<table>
<thead>
<tr>
<th></th>
<th>Debian Sid</th>
<th>Ubuntu Precise</th>
</tr>
</thead>
<tbody>
<tr>
<td>Priority: required</td>
<td>37</td>
<td>70</td>
</tr>
<tr>
<td>Essential: yes</td>
<td>25</td>
<td>24</td>
</tr>
<tr>
<td>Build-Essential: yes</td>
<td>11</td>
<td>44</td>
</tr>
<tr>
<td>how many binary packages</td>
<td>106</td>
<td>140</td>
</tr>
<tr>
<td>how many source packages</td>
<td>55</td>
<td>75</td>
</tr>
</tbody>
</table>

Many packages in XC would cross-build just fine if their cross-build-dependencies could be resolved using Multi-Arch.

Challenge N. 1: Automatically Cross compile the minimal build system.
Test cross-build-dependency resolution

With `apt-get` (adding an armel as foreign architecture):

```bash
apt-get --simulate --host-architecture=armel build-dep <package>
```

Or with `dose-buildebcheck` (static check):

```bash
dose-buildebcheck --success --failures --explain \
    --deb-native-arch=amd64 \
    --deb-host-arch=armhf \
    ubuntu_dists_quantal_main_binary-amd64_Packages \
    ubuntu_dists_quantal_main_binary-armhf_Packages \
    ubuntu_dists_quantal_main_source_Sources
```
We can’t cross compile the minimal build system (yet !)

Here is a table of the currently unsatisfied cross-build-dependencies:

<table>
<thead>
<tr>
<th>unsatisfied cross-build-dependency</th>
<th>source packages failing because of it</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcl-dev</td>
<td>db</td>
</tr>
<tr>
<td>autoconf</td>
<td>acl, attr, binutils, gdbm, libsigsegv, make-dfsg, shadow, slang2, tar</td>
</tr>
<tr>
<td>texlive-latex-base</td>
<td>bash, mpfr4</td>
</tr>
<tr>
<td>python</td>
<td>bsdmainutils, build-essential, file, glib2.0, linux</td>
</tr>
<tr>
<td>dh-buildinfo</td>
<td>coreutils</td>
</tr>
<tr>
<td>po-debconf</td>
<td>dash, insserv. sysvinit, util-linux</td>
</tr>
<tr>
<td>texi2html</td>
<td>diffutils, e2fsprogs</td>
</tr>
<tr>
<td>libtimedate-perl</td>
<td>dpkg</td>
</tr>
<tr>
<td>perl-modules</td>
<td>eglibc, gettext, libtext-charwidth-perl, libtext-icnv-perl, libxml2, xz-utils</td>
</tr>
<tr>
<td>dejagnu</td>
<td>findutils, libffi</td>
</tr>
<tr>
<td>locales</td>
<td>gawk</td>
</tr>
<tr>
<td>gsfonts-x11</td>
<td>gcc-4.7</td>
</tr>
<tr>
<td>libgcj-common</td>
<td>gcc-defaults</td>
</tr>
<tr>
<td>mingw-w64</td>
<td>gzip</td>
</tr>
<tr>
<td>gem2deb</td>
<td>libselinux, libsemanage</td>
</tr>
<tr>
<td>docbook-xml</td>
<td>pam</td>
</tr>
<tr>
<td>netbase</td>
<td>perl</td>
</tr>
</tbody>
</table>
Which packages can be natively compiled from XC?

Maximal set of source package that can be compiled natively.

- $B_i$: set of binary packages that are currently available.
- $S$: set of packages that we want to compile.
- $S_i$: set of source packages that can be successfully compiled.

1: procedure $\text{BUILD}(S_i, B_i, S)$
2: $S_{i+1} \leftarrow \text{find\_installable}(B_i, S)$
3: if $S_{i+1} = \emptyset$ then
4: return $S_i$
5: else
6: $B_{i+1} \leftarrow \text{Bin}(S_{i+1}) \cup B_i$
7: return $\text{BUILD}(S_i \cup S_{i+1}, B_{i+1}, S \setminus S_{i+1})$
8: $\text{ALLNATIVE} \leftarrow \text{BUILD}(\emptyset, \text{Bin}(XC), S)$
**The dependency graph**

- **Two types of vertex**
  - source packages.
  - build-dependency set (binaries needed to build a source package)

- **Two types of edges**
  - build-dep (source $\rightarrow$ binary)
  - build-from (binary $\rightarrow$ source)

- Built iteratively by adding connecting each source package to the set of its build dependencies and each build dependencies set to all source packages whose binaries are build from.

- Packages that are cross-built ($p \in XC$) or with Architecture:all are excluded from the dependency graph.
The control fields Build-Depends-Indep and Build-Conflicts-Indep identify dependencies or conflicts for building architecture:all packages.

We are not interested to build architecture:all packages therefore we can drop Build-Depends-Indep and Build-Conflicts-Indep dependencies.

Find Weak dependencies:

- Manually identify packages that are not strictly needed to compile a working, albeit not full, package.
- Use external information to identify weak packages (gentoo compile flags).
- Add **build profiles** (ex. stage1, embedded, nodoc, etc) to source packages (more later about build profiles).
Some numbers on the build graph

- The dependency graph generated for Debian Sid has 39486 vertices.
- It has only one central SCC with 1027 vertices.
- Eight other SCC with 2 to 7 vertices.
- Contains not-nice packages like: nautilus, iceweasel, metacity, evolution, etc.
- Contains many build dependency cycles.
- Challenge N. 3 (Automatically) Remove build dependencies.
**Dependency graph analysis**

We can easily identify:

- binary/source nodes with most/least incoming/outgoing edges
- most/least connected nodes
- source packages only missing a few build dependencies
- binary packages with highest ratio of source packages it needs to be built and source packages that build depend on it

- source packages with highest ratio of build dependencies and source packages that build-depend on packages that depend on it
Current Unresolved Issue in Debian

- Provide a build order is still difficult because:
  - unsatisfied cross build dependencies because of missing multi-arch annotation
  - insufficient number of reduced build dependencies to solve dependency cycles
- what is blocking the above:
  - wanna-build doesn’t support architecture qualifiers (pkg:any, pkg:native, pkg:amd64, …)
  - no decision on format of reduced build dependencies
- after both issues are solved, changes have to be manually implemented into actual packages
Future work

- Identify a list of plausible weak dependencies (Work in progress to use Gentoo build-flags)
- Devise an algorithm to automatically break build cycles using weak dependencies (almost done)
- Create a topological sort of the build dependency graph (almost done)
- Provide a build order to be used to bootstrap debian of a foreign architecture.
- Generalize this solution to a larger class of problems.
Tools and Resources

All our tools and experiments are available:

- Debian Bootstrap:
  https://gitorious.org/debian-bootstrap/bootstrap
- Dose: https://gforge.inria.fr/projects/dose/
- dose-builddebcheck:
  http://packages.debian.org/wheezy/dose-builddebcheck
- Main page: http://wiki.debian.org/DebianBootstrap
- Lots of details:
  http://wiki.debian.org/DebianBootstrap/TODO
- Multi-Arch Cross spec
  https://wiki.ubuntu.com/MultiarchCross
- Multi-Arch spec: https://wiki.ubuntu.com/MultiarchSpec