Bootstrapping Debian (or derivatives) for a new architecture

Johannes Schauer

Jacobs University Bremen

20-09-2012

Overview

- Google Summer of Code 2012 project
- Mentored by wookey and Pietro Abate
- written in OCaml and Python
- using dose3
- released under GNU LGPL3+
- `https://gitorious.org/debian-bootstrap/bootstrap`
- IRC: `#debian-bootstrap @ irc.debian.org`
- mailing list: `debian-bootstrap@lists.mister-muffin.de`
- please ask questions any time

Motivation

- Debian packages are neither made to be cross compilable nor to be built without an existing full Debian (no stage builds)
- but for each new port a set of source packages has to be cross compiled and/or built with reduced build dependencies
- because there is no official procedure for above tasks, the process is long and manual, involves foreign distributions, manual hacking of packages
- Debian was ported to more than 20 architectures so the process is executed roughly once per year

Motivation (cont.)

- porting Debian to a new architecture would less time consuming and less problematic
- remove the need of other distributions during porting, make Debian more universal as it will then be able to build itself from scratch
- update lagging architectures
- build Debian for architectures that cannot build themselves (avr32)
- make sub-arch builds optimized for a specific CPU easier

Outline

Cross compiling the initial set of packages

- preference of native compilation over cross compilation
- when starting with nothing a minimal build system has to be cross compiled

Finding the minimal set of source packages that must be cross compiled

1. find binary packages that are either:
   - `Essential:Yes`
   - `Build-Essential:yes`
   - `Priority:Required`
   - `Package:build-essential`
   - most likely also debhelper as 79% of the archive depend on it
2. get their installation set
3. find the source packages that build the installation set excluding `Architecture:all` packages

Start building natively

1. based upon minimal build system
2. install as many binary packages as possible
   - all `Architecture:all` packages
   - binary packages that were cross compiled
   - binary packages that were natively compiled in earlier iterations
3. build as many source packages as possible, if some new source packages could be built, go to 2
4. investigate situation

Investigate build dependency cycle situation

- if there are still packages left after the algorithm finishes, then there are circular build dependencies
- try to solve circular dependency situation by looking closer at
  - binary packages needed as build dependency by most source packages
  - smallest dependency cycles
  - source packages with least build dependencies missing
  - source packages with only "weak" build dependencies missing

Availability of binary packages

- if a binary package is not yet available in the new system, then this is because of one of the following reasons:
  - its source package cannot be built
  - binary packages from its installation set are not available
  - both of the above
- a binary package can be made available by:
  - cross compiling the source packages(s) that provide it and/or its installation set
  - introducing reduced build dependencies
  - moving build dependencies into `Build-Depends-Indep`
  - a combination of the above

Making a package available through cross compilation

1. find the installation set of the selected binary package
2. remove all binary packages that are already available
3. find the source packages that build the rest

- but native compilation is preferred
- but also blocked by build dependency cycles
- for finding reduced build dependencies that break the build dependency cycles, the dependency graph has to be built

The dependency graph

- the dependency graph has two types of vertices
  - source package nodes
  - binary packages and their installation set
- the dependency graph has two types of edges
  - build dependency (source $\rightarrow$ binary)
  - builds-from (binary $\rightarrow$ source)

The dependency graph (cont.)

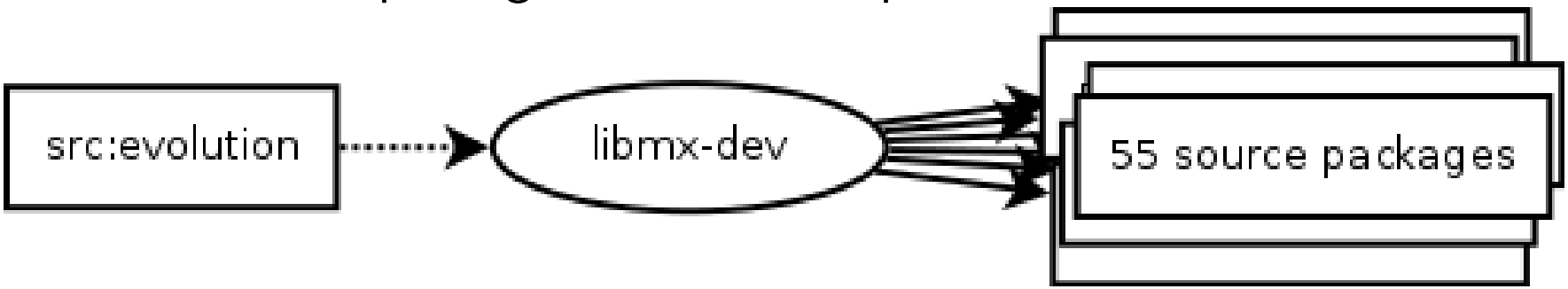
- graph is built by recursively adding nodes and edges, starting from the root node that is the investigated package
- source package nodes connect to all binary package node that represent their build dependencies
  - except for binary packages that are already installable
- binary packages (and their installation set) connect to all source package nodes that they build from
  - installation set excludes binary packages that are already available
    - natively built
    - cross built
    - `Architecture:all`

Possibilities of dependency graph analysis

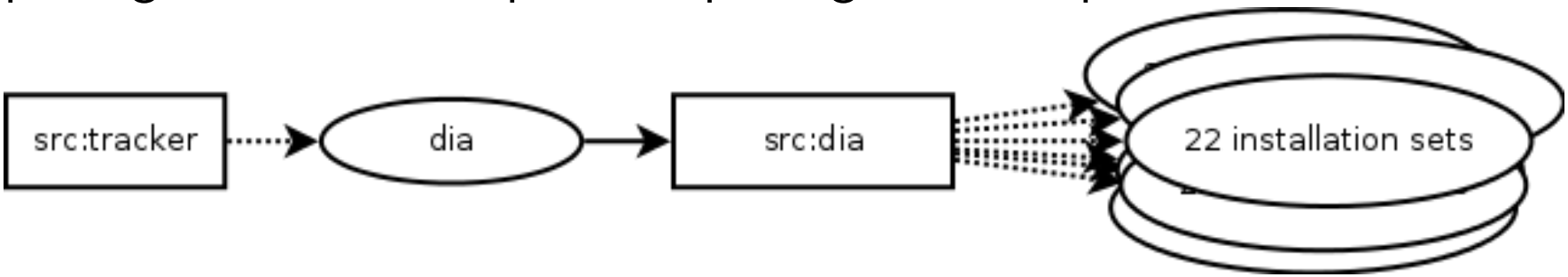
- show statistics
  - binary/source nodes with most/least incoming/outgoing edges
  - most/lesat connected nodes
  - highest/lowest ratios
- show/save DOT graph
- show cycles up to length N (N is even)

Possibilities of dependency graph analysis (cont.)

- 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



Enumerating elementary circuits of a directed graph

- using Johnson's algorithm (1975)
- lack of existing, well-tested cycle enumeration code
- validated by comparing output with implementations of Johnsons's algorithm in Java, Tarjan's algorithm in Python and Hawick and James' algorithm in D
- added functionality to only enumerate cycles up to a certain length and amount
- code at `https://github.com/josch/cycle_test`

Distribution graphs

- observation: most packages are involved in the same SCC
- do not build a graph starting from package X but for a whole distribution by same principles as above
- finding cycles in the resulting graph takes long (4 hours on 2.5GHz Core i5, 700MB memory consumption)
- many packages exist in the graph that are not part of the "core" packages but are just leave nodes

Reduced distribution

- idea: start analysis on a "reduced distribution" and continue from there
- what is a reduced distribution:
  - contains a set of source packages A and a set of binary packages B
  - all binary packages in B except for `Architecture:all` packages can be built from the source packages in A
  - all source packages in A are buildable with the binary packages in B
- analysis of a reduced distribution only takes minutes and results were shown to be the same as with a full distribution

Reduced distribution (cont.)

- how the package selection is made:
  1. start with initial selection of binary packages (eg: essential plus build-essential)
  2. find source packages that build those binary packages
  3. find binary packages that are necessary to build those source packages, if those are more than durign the last iteration, go to 2.
  4. done

Deducing a build order

1. starting from zero
2. cross compile all source packages from the minimal set of packages that can be cross compiled, if done, go to 4.
3. use reduced build dependencies to solve a dependency cycle and go to 2.
4. switch to native compilation
5. build all binary packages that can be built, if all binary package can be built, go to 7.
6. use reduced build dependencies and cross compilation to solve a dependency cycle and go to 5.
7. done

Deducing a build order (cont.)

- not possible yet because of:
  - unsatisfied cross build dependencies because of missing multiarch annotation
  - insufficient number of reduced build dependencies to solve dependency cycles
- what is blocking the above:
  - wanna-build doesnt 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 implemented into actual packages

Build profiles

- current proposal for reduced build dependencies: build profiles
- `Build-Depends: foo [i386 arm] <!stage1 embedded>`
- format similar to architecture specifiers
- has other uses like building a source package for embedded architectures or without docs
- trivial to implement in dpkg and dose3 (patches available for both)
- no duplication of build dependency list (less bitrot)

Results

- actual output (package listings etc) can be seen at `http://wiki.debian.org/DebianBootstrap/TODO`
- list of source packages for a minimal build system
- list of source packages with their unsatisfied cross build dependencies
- packages depending on `Multi-Arch:foreign` packages without `:any` qualifier
- list of packages that are `Multi-Arch:none` in Debian but not in Ubuntu
- packages that build `Architecture:all` packages but have no `Build-Depends-Indep` field but a `binary-indep` and/or `build-indep` target

Results (cont.)

- list of source packages that are part of the main scc
  - 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...

Results (cont.)

- list of source packages only missing a few build dependencies
- list of source packages that build-depend on many others but are only needed by few binary packages which are in turn only needed buy a few source packages
- list of source packages that are only needed by few source package but need many other source packages to be built to satisfy their runtime dependencies
- list of dependency cycles of length 2 and 4
  - example of cycles of length 2 (source package build depends on the binary package it builds): vala, python, mlton, fpc, sbcl, ghc
  - no more dependency cycles because there would be 958 cycles of length 6, 5566 cycles of length 8 and 37839 cycles of length 10

Minimal build system

	Debian Sid	Ubuntu Precise
`priority:required`	37	70
`essential:true`	25	24
`buildessential:true`	11	44
the above plus dependencies	106	140
number of source packages	55	75

Reduced distribution

	Debian Sid	Ubuntu Precise
src/bin in original repositories	18266/37781	3305/8076
src/bin without important	645/2324	522/1838
src/bin with important	679/2403	541/1871
src/bin imp. + task-gnome-desktop	855/2853	-
src/bin imp. + ubuntu-desktop	-	718/2467
src/bin imp. + task-kde-desktop	791/2769	-
src/bin imp. + kubuntu-desktop	-	618/2158

Cross everything

- if there were no reduced build dependencies but only cross compilation, crosscompiling the following number of packages would make the whole archive available

	task-gnome-desktop	Precise	ubuntu-desktop
to cross	158	176	157

- it cannot yet be shown that reduced build dependencies are unavoidable
- they are unavoidable if they are needed to build the minimal set of packages

Future

- allow to analyze the dependency graph for cross building
- find more clever methods to break the main scc
- combine everything for a final buildorder
- close work with wookey to make base packages crosscompile
- work is done based on Ubuntu because of wanna-build blocker
- make use of Gentoo USE flags

Questions?