Rebuilding desktop distributions for small devices: Handhelds Mojo

Andrew Christian
Nokia Research Center, Cambridge US
The problem...

Mobile device
- Maemo Linux, ~700 packages
- Scratchbox build environment

Development Laptop
- Ubuntu Gutsy Linux, ~12,000 packages
- Native build environment

The mobile device has a limited “off-the-shelf” environment
What we’d like

A distribution for mobile & embedded devices with:

- Large numbers of up-to-date binary packages
- Well-defined releases with security and bug fixes
- Code that takes full advantage of the processor
- Easily interoperates with the developer’s desktop

We don’t want to spend a lot of time building and maintaining this...
Why existing solutions fall short...

Debian
- **Pro:** Large number of packages (>10,000)
- **Con:** Not optimized for hardware, infrequent stable releases

Open Embedded
- **Pro:** Good optimization for hardware, interesting GUI work
- **Con:** Small number of packages (~1680), doesn’t match desktop

Maemo
- **Pro:** Good optimization for hardware, GUI
- **Con:** Small number of packages (~700), Scratchbox can be tricky, _really_ doesn’t match desktop
The Mojo approach

Strategy
1. Build standard desktop distributions for small devices
2. Modify the *minimum* number of packages necessary to compile
3. Compile each distribution once for each hardware architecture

Start with
- Ubuntu distributions and updates
- Latest ARM instructions set
# Mojo distribution naming scheme

<table>
<thead>
<tr>
<th>Ubuntu</th>
<th>Mojo</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.04 Feisty Fawn</td>
<td>Frisky Firedrake</td>
</tr>
<tr>
<td>7.10 Gutsy Gibbon</td>
<td>Grumpy Griffin</td>
</tr>
<tr>
<td>8.04 Hardy Heron</td>
<td>Hasty Hippogriff</td>
</tr>
</tbody>
</table>

In the future we’d like to extend this to Debian and other distributions.
The rest of the talk...

Critical choices and challenges

- The build process – getting a stable place to stand
- Matching the toolchain
- Build machines – handling the “native” problem
- Naming of names – Debian architecture

Current status

- State of the distributions
- Using the distributions
Desktop distribution build process

Key points

1. The build system is running its own packages. Iteration required!
2. The build system runs on *native* hardware
3. The toolchain is intrinsic to the distribution and gets compiled along with all of the other packages
Challenge #1: A stable place to stand

A Debian-style build system is a moving target

• The build system relies on having a large number of installed binary packages
• The binary packages have to be (mostly) compatible with what you are building
• The system is inherently incremental: you build packages, install them, build the next set, install them, ....

Where can we start? (A classic “chicken-and-egg” problem)

The first challenge: EABI

Mojo: Handhelds Rebuild Project
EABI vs. OABI

Changes in the ARM Application Binary Interface

- Floating point handling
- Structure alignment
- New Linux syscall interface (can co-exist with old)

Supported by:

- ARMv4T and higher (ARMv4 with some hacks)
- gcc 4.1.0 (4.1.1 for ARMv4T), binutils 2.16.92, glibc 2.4
- Linux kernel 2.6.16+

EABI and OABI do not interoperate
Building a distribution on EABI

First, you need an EABI distribution!
- Debian “ARM” = OABI
- Debian “ARMEL” & “ARMEB” = EABI

Early in 2007 ADS released a version of Debian compiled with EABI
- Generated from an Open Embedded EABI distribution

First pass on Debian ARM machine with ADS-based chroot image
Challenge #2: Matching the toolchain

A toolchain is the combination of:

- C compiler (gcc)
- Linking and object tools (binutils)
- Standard C libraries (glibc)

A “good” toolchain is one that passes a most of its test suites.

- ARM is not the most popular architecture: building a “good” ARM toolchain requires a fair bit of testing and patching
- Toolchains depend in surprising ways on all sorts of other packages (e.g. Perl, bash, …)
- Number of errors from test suite decreases as you iterate; for example, for gcc 4.1.2, we went from 11 to 5 to 0 with each iteration.
The toolchain in Ubuntu

<table>
<thead>
<tr>
<th></th>
<th>gcc</th>
<th>binutils</th>
<th>libc6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dapper</td>
<td>4.0.3-1</td>
<td>2.16.1.cvs2006...</td>
<td>2.3.6-0ubuntu20</td>
</tr>
<tr>
<td>Edgy</td>
<td>4.1.1-6ubuntu3</td>
<td>2.17-1ubuntu1</td>
<td>2.4-1ubuntu12</td>
</tr>
<tr>
<td>Feisty</td>
<td>4.1.2-1ubuntu1</td>
<td>2.17.20070103...</td>
<td>2.5-0ubuntu4</td>
</tr>
<tr>
<td>Gutsy</td>
<td>4.1.2-9ubuntu2</td>
<td>2.18-0ubuntu3</td>
<td>2.6.1-1ubuntu9</td>
</tr>
<tr>
<td>Hardy</td>
<td>4.2.3-1ubuntu3</td>
<td>2.18.1~cvs2008...</td>
<td>2.7-10ubuntu3</td>
</tr>
</tbody>
</table>

Native ARM toolchains can be a bit of a problem...

- glibc <= 2.5 and binutils <= 2.17 had ARM C++ errors
- A surprisingly large number of packages affect the toolchain
- EABI supported by: gcc 4.1.0 (4.1.1 for ARMv4T), binutils 2.16.92, glibc 2.4
Bootstrapping from ADS Debian Etch

ADS ARMEL packages

Frisky revision 1

Frisky revision 2

Frisky revision 3

Many iterations

A few iterations

One iteration

ADS ARMEL packages

Frisky revision 1

Frisky revision 2

Frisky revision 3

gcc

binutils

glibc

Many iterations

A few iterations

One iteration

ADS ARMEL packages

Frisky revision 1

Frisky revision 2

Frisky revision 3

gcc

binutils

glibc

Many iterations

A few iterations

One iteration
Challenge #3: Handling the “native” problem

Desktop distributions are not cross-built: you need an ARM-based machine to build an ARM-based distribution

- Option #1: Fundamentally change the build system using something like Scratchbox. We couldn’t find a good way to do this without a lot of source package modifications
- Option #2: Create a build cluster of ARM-based machines.

Remember: One goal is the absolute minimal number of modifications to existing source packages
## Options for “native” build machines

<table>
<thead>
<tr>
<th>Pure ARM</th>
<th>QEMU-ARM Chroot</th>
<th>QEMU-SYSTEM-ARM</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM Distribution</td>
<td>ARM Distribution</td>
<td>ARM Distribution</td>
</tr>
<tr>
<td>ARM Kernel</td>
<td>ARM Chroot</td>
<td>ARM Kernel</td>
</tr>
<tr>
<td>ARM Hardware</td>
<td>QEMU-ARM</td>
<td>Virtual ARM Hardware</td>
</tr>
<tr>
<td>x86 Distribution</td>
<td>x86 Distribution</td>
<td>QEMU-SYSTEM-ARM</td>
</tr>
<tr>
<td>x86 Kernel + <code>binfmt</code></td>
<td>x86 Kernel</td>
<td>x86 Distribution</td>
</tr>
<tr>
<td>x86 Hardware</td>
<td>x86 Hardware</td>
<td>x86 Hardware</td>
</tr>
</tbody>
</table>

In early 2007 we looked at the time and cost to build a sufficiently fast cluster
2007 cluster: Native ARM build machines

20 home-built 1U ARM boxes
- 600 MHz Intel 80219 (ARMv5)
- 256 MB DRAM / 160 GB disk
- Ethernet, USB
- 593 BogoMIPS
- gcc-4.1 compile and test suite: 32 hours

4 days to build
Frisky Main
2008 cluster: QEMU virtual ARM build machines

5 Dell PWS 390 (10 virtual machines)
- 2.66 GHz Intel Core2
- 2 GB DRAM / 80 GB disk
- QEMU 0.9.1, Versatile PB
- 650 BogoMIPS
- gcc-4.1 compile test: 25 hours

25% faster machines than original cluster
Challenge #4: Debian architecture names

Debian ARM architecture schemes

- **arm**: ARMv3 + hard float, `package.arm.deb`
- **armel**: ARMv4T, EABI, little-endian, `package.armel.deb`
- **armeb**: ARMv4T, EABI, big-endian, `package.armeb.deb`

The “arm/armel/armeb” architecture information appears in the `Architecture` field of the Debian control file. Changing means changing every source file....

We’d like to optimize our code for the *exact* processor type, not a generic one.
## Solutions to the naming problem

### Option #1: Add new architectures

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>armv5el</td>
<td>ARMv5, EABI, little-endian (soft float)</td>
</tr>
<tr>
<td>armv5teb-hard</td>
<td>ARMv5, thumb, EABI, big-endian, hard float</td>
</tr>
<tr>
<td>armv6elvfp</td>
<td>ARMv6, EABI, little-endian, vector floating point</td>
</tr>
</tbody>
</table>

This requires modifying *each* source package once for *every* architecture we compile.

### Option #2: Don’t follow Debian model...
Our solution: Differentiate by feed

CLASSIC  / ubuntu / dists / feisty / main / binary-i386, binary-arm, binary-sparc, source ...
       / universe / binary-i386, ....
       / feisty-updates / main / binary-i386...
       / gutsy...
       / pool...

MOJO     / frisky-armv5el / dists / frisky / main / binary-arm, source
       / universe...
       / frisky-updates...
       / pool...
       / frisky-armv6el-vfp / dists / frisky / main / binary-arm, source
       / universe...
       / frisky-source / dists / frisky / main / source
The implications of differentiating by feed

- No source packages need to be changed – we just use the “arm” architecture
- Debian systems use the default settings of the toolchain – so we need to modify the toolchain once for each architecture target
- The source packages end up in three different directories:
  1. Replicated copy from original distribution
  2. Common directory of modified source packages ("frisky-source")
  3. Architecture-specific directory ("frisky-armv5el")
- We’re acting against explicit Debian policy. This is a subject for discussion with Debian. Is there a better solution?
Where are we?

Critical choices and challenges

- The build process – getting a stable place to stand
- Matching the toolchain
- Build machines – handling the “native” problem
- Naming of names – Debian architecture

Current status

- State of the distributions
- Using the distributions
Current state

frisky-armv5el
• Main, Universe “largely” complete and stable
• Updates and security in progress

frisky-armv6el-vfp
• Compiling Main

grumpy-armv5el
• Main (first round) complete
# Frisky: What is “largely” complete?

<table>
<thead>
<tr>
<th></th>
<th>Feisty source packages</th>
<th>Modified source packages</th>
<th>Feisty binary-i386 packages</th>
<th>Frisky binary-arm packages</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>2768</td>
<td>61</td>
<td>5099</td>
<td>4265 (85%)</td>
</tr>
<tr>
<td>restricted</td>
<td>5</td>
<td>0</td>
<td>33</td>
<td>0</td>
</tr>
<tr>
<td>universe</td>
<td>9596</td>
<td>1</td>
<td>15642</td>
<td>12081 (77%)</td>
</tr>
<tr>
<td>multiverse</td>
<td>399</td>
<td>0</td>
<td>595</td>
<td>0</td>
</tr>
</tbody>
</table>

Build time is ~4 days on native ARM cluster for Main, ~10 days for Universe
What happened to the source packages?

Component “Main”

- Fully built: 2120
- Partially built: 323
- Failed to build: 49
- Wrong Architecture: 276

April 2008
Mojo: Handhelds Rebuild Project
What have we modified?

Added one package

- handhelds-keyring: For package signatures

Modified 61 packages:

- Most are just a few lines of code fixing dependencies or ARM-specific bugs
- Five packages (tar, tzdata, gzip, coreutils, docbook2x) pulled from later distributions to match glibc2.6
- A few larger patches to work around ARM issues. E.g., qt-x11-free XML parsing bug needed removal of '\n\r' at end of .ui files.
What packages haven’t built?

- The ARM machines have trouble with large C++ libraries. GCC can crash on the linking stage with an out-of-memory error (KDE is a particular challenge)
- We don’t have a Java or Mono for ARM
- A number of math libraries depend on the g77 Fortran compiler
- Documentation packages (they have remarkable dependencies)

It’s a bit of a hobby to continue to patch and fix packages to fill out the distribution
How can you try this out?

Option #1: Put it on your desktop in a virtual machine

• A pre-built file system is available and works with the QEMU VersatilePB emulator
• The netboot installer “mostly” works and will allow a remote installation of Frisky onto a clean filesystem.

Option #2: Use it on an existing device

• N800 demonstration
Final thoughts: What we’re doing now

- Automating the security and bug-fix feeds
- Patching source packages that failed to build
- Submitting patches back to Debian and Ubuntu
- Starting up new distributions
- Filling out the architecture
- Fixing up the installers
- ...and using these distributions, of course...

http://mojo.handhelds.org