Virtual perspectives on cross-compiling

Alex Bennée

Linaro Connect BUD17-402
Who Am I

- Alex Bennée
- Senior Virtualisation Engineer
- @ Linaro since 2013
- Mostly work on QEMU's TCG
Virtualisation Solves Everything!

PROBLEM COMPILING?

VM ALL THE THINGS!
Talk Structure
Use Case - Kernel Compile
The Kernel Compile

Running:

```bash
export PATH=${HOME}/${COMPILER}/bin:$PATH
export ARCH=arm
export CONFIG_CROSS_COMPILE=arm-linux-gnueabihf-
make -j9 | tail -n 3
```

Gives:

```bash
LD     arch/arm/boot/compressed/vmlinux
OBJCOPY arch/arm/boot/zImage
Kernel: arch/arm/boot/zImage is ready
```
Kernel Compiles Are Easy!

- Self Contained
- No external libraries
  - no headers
  - no shared objects
  - no static archives
  - no pkg-config
Use Case - Simple Test Case
The Code

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    printf("Hello World\n");
}
```
The Compile

Running:

```
aarch64-linux-gnu-gcc --static simple-hello.c -o simple-hello
file simple-hello
```

Gives:

```
simple-hello: ELF 64-bit LSB executable, ARM aarch64, version 1 (SYSV), statically linked, for GNU/Linux 3.7.0, BuildID[sha1]=4712144fd1986c295641da66f7f5d40d35a2b1fb, not stripped
```
Use Case - Using other libraries
The Code

```c
#include <stdio.h>
#include <glib.h>

int main(int argc, char **argv)
{
    g_message("Hello World\n");
}
```
A Native Compile

Getting include paths:

```
pkg-config glib-2.0 --cflags --libs

-I/usr/include/glib-2.0 -I/usr/lib/x86_64-linux-gnu/glib-2.0/include -lglib-2.0
```

Compiling:

```
gcc glib-hello.c ${FLAGS} -o glib-hello-native
```

Gives:

```
glib-hello-native: ELF 64-bit LSB executable, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32,
BuildID[sha1]=1a4db3259cd31f564d2ec60654ef478d372b20d2, not stripped
```
A Cross Compile

Running:

FLAGS="$(pkg-config glib-2.0 --cflags --libs)"
aarch64-linux-gnu-gcc glib-hello.c ${FLAGS} -o glib-hello-arm64

Gives:

/usr/lib/gcc-cross/aarch64-linux-gnu/5/../../../../aarch64-linux-gnu/bin/ld: cannot find -lglib-2.0
collect2: error: ld returned 1 exit status
Use Case - Multi-arch user-space build
Enabling ARM64 on Debian

dpkg --add-architecture arm64
apt-get update -qq
apt-get install -yy libglib2.0-dev:arm64
apt-get build-dep -a arm64 qemu
Compile Flags

Call:

```
aarch64-linux-gnu-pkg-config glib-2.0 --cflags --libs
```

For:

```
-I/usr/include/glib-2.0 -I/usr/lib/aarch64-linux-gnu/glib-2.0/include -L/usr/lib/aarch64-linux-gnu -lglib-2.0
```
Compile

Running this:

```
aarch64-linux-gnu-gcc glib-hello.c ${FLAGS} -o glib-hello-arm64-pkgconfig
file glib-hello-arm64-pkgconfig
```

Gives:

```
glib-hello-arm64-pkgconfig: ELF 64-bit LSB executable, ARM aarch64, version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux-aarch64.so.1, for GNU/Linux 3.7.0, BuildID[sha1]=cf29dbefb15b5c9936760e765f313e0ba4c3399a, not stripped
```
Multi-Arch Pros/Cons
Sing Debian's Praises

- 10 supported architectures
- Many additional "ports"
- Excellent maintainers ;-)
Caveats

- Secondary architecture must match versions
  - even minor difference will fail
- Not all packages are multi-arch "clean"
  - ARM is pretty good, but not complete
- Very Debian
  - Even Ubuntu needs ports
  - Not portable to other
Other Options

- Chroot
- Cross-build system
  - Buildroot
  - OpenEmedded
Chroot Issues

- Often project specific
- Unknown provenience
- Can be fiddly to set-up
Build Systems

- Builds full-stack
  - Toolchain
  - Libraries
  - Project Manifest
- Designed as Distro builders
- Massive overkill
Our Wishlist

- Concisely defined build-environment
- Widely available across distros
- Fully buzzword compliant
Docker
FROM debian:stable-slim
# Setup Emdebian
RUN echo "deb http://emdebian.org/tools/debian/ jessie main" >> /etc/apt/sources.list
RUN curl http://emdebian.org/tools/debian/emdebian-toolchain-archive.key | apt-key add -
# Duplicate deb line as deb-src
RUN cat /etc/apt/sources.list | sed "s/deb/deb-src/" >> /etc/apt/sources.list
# Add arm64multiarch
RUN dpkg --add-architecture arm64
# Install what we need
RUN apt update
RUN apt install -yy crossbuild-essential-arm64
RUN apt-get build-dep -yy -a arm64 qemu
Docker guest notes

• Defaults to root

```
RUN id alex 2>/dev/null || useradd -u 1000 -U alex
```

• Self-contained file-system
Invocation

Running:

docker run --rm \ 
  --user=$(id -un) -v $(pwd):$(pwd) -w $(pwd) \ 
  qemu:debian-arm64-cross \ 
  aarch64-linux-gnu-gcc glib-hello.c \ 
  -I/usr/include/glib-2.0 \ 
  -I/usr/lib/aarch64-linux-gnu/glib-2.0/include \ 
  -L/usr/lib/aarch64-linux-gnu -l/lib/arm64v8a-linux-gnu \ 
  -o glib-hello-arm64-docker

Gives:

glib-hello-arm64-docker: ELF 64-bit LSB executable, ARM aarch64, version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux-aarch64.so.1, for GNU/Linux 3.7.0, BuildID[sha1]=cf29dbefb15b5c9936760e765f313e0ba4c3399a, not stripped
Use Case - When Multi-Arch doesn't work

- Multi-Arch might be broken
- Development OS may not be Debian
If only

IF ONLY....

...I HAD SOME SORT OF CROSS-ARCHITECTURE SIMULATOR
Guest thinks it is an ARM system, with complete ARM rootfs
Caveats

- `binfmt_misc` not containerised
- static QEMU safer
- slower than native cross
Use Case - QEMU Build System
Why?

• Portable build recipes
  ▪ used by Patchew
• Building guest test cases
Build QEMU and run tests inside Docker containers

Available targets:

- docker: Print this help.
- docker-test: Run all image/test combinations.
- docker-clean: Kill and remove residual docker testing containers.
- docker-TEST@IMAGE: Run "TEST" in container "IMAGE".
  Note: "TEST" is one of the listed test name,
  or a script name under $QEMU_SRC/tests/docker/;
  "IMAGE" is one of the listed container name.
- docker-image: Build all images.
- docker-image-IMAGE: Build image "IMAGE".
- docker-run: For manually running a "TEST" with "IMAGE"

Available container images:
  gentoo travis fedora min-glib debian-s390x-cross centos6
  ubuntu debian-bootstrap debian-arm64-cross debian debian-armhf-
Available tests:
   test-clang test-mingw test-build test-quick test-full

Available tools:
   travis

Special variables:
   TARGET_LIST=a,b,c  Override target list in builds.
   EXTRA_CONFIGURE_OPTS="..."  Extra configure options.
   IMAGES="a b c ..":  Filters which images to build or run.
   TESTS="x y z ..":  Filters which tests to run (for docker-test).
   J=[0..9]*  Overrides the -jN parameter for make commands
               (default is 1)
   DEBUG=1  Stop and drop to shell in the created container
   NOUSER  Define to disable adding current user to containers passwd.
   NOCACHE=1  Ignore cache when build images.
   EXECUTABLE=<path>  Include executable in image.
Examples

make docker-test-build@centos6 J=9
make docker-test-quick@travis J=9
make docker-test-build@debian-arm64-cross TARGET_LIST="arm-softmmu,arm-linux-user" J=9
make docker-test-build@debian-ppc-user TARGET_LIST="ppc-softmmu" J=9
Summary

- Kernel compiles are the simple case
- User-space adds complexity
- Multi-Arch helps with cross-compiling
- Docker allows us to paper over the cracks
- Combine Docker with QEMU for double the fun
Questions?
Thank You

#BUD17

For further information: www.linaro.org

BUD17 keynotes and videos on: connect.linaro.org
Extra Slides
#!/bin/sh
# Simple wrapper for debootstrap, run in the docker build context
FAKEROOT=`which fakeroot 2> /dev/null`
# debootstrap < 1.0.67 generates empty sources.list, see Debian#732255
MIN_DEBOOOSTRAP_VERSION=1.0.67
if [ -z $FAKEROOT ]; then
...
if [ -z "${DEB_ARCH}" ]; then
...
if [ -z "${DEB_TYPE}" ]; then
...
# Check for debootstrap or checkout debootstrap from its
# upstream SCM and run it from there.
...
# Finally check to see binfmt_misc is setup
BINFMT_DIR=/proc/sys/fs/binfmt_misc
if [ ! -e $BINFMT_DIR ]; then
...
$[FAKEROOT] $[DEBOOOSTRAP] --variant=buildd --foreign --
arch=${DEB_ARCH} $DEB_TYPE . http://httpredir.debian.org/debian ||
exit 1
exit 0
# Create Debian Bootstrap Image
#
# This is intended to be pre-poluated by:
#  - a first stage debootstrap (see debian-bootstrap.pre)
#  - a native qemu-$arch that binfmt_misc will run
FROM scratch

# Add everything from the context into the container
ADD . /

# Patch all mounts as docker already has stuff set up
RUN sed -i 's/in_target mount/echo not for docker in_target mount/g' /debootstrap/functions

# Run stage 2
RUN /debootstrap/debootstrap --second-stage

# At this point we can install additional packages if we want
# Duplicate deb line as deb-src
RUN cat /etc/apt/sources.list | sed "s/deb/deb-src/" >> /etc/apt/sources.list
RUN apt-get update
RUN apt-get -y build-dep qemu