Developing for non-X86 targets using QEMU

Rob Landley and Mark Miller
Impact Linux, LLC
http://impactlinux.com

Introduction

Emulation allows even casual hobbyist developers to build and test the software they write on multiple hardware platforms from the comfort of their own laptop.

QEMU is rapidly becoming a category killer in open source emulation software, capable of not only booting a Knoppix CD in a window but booting Linux systems built for ARM, MIPS, PowerPC, SPARC, sh4, and more.

This talk covers application vs system emulation, native vs cross compiling (and combining the two with distcc), using QEMU, setting up an emulated development environment, real world scalability issues, using the Amazon EC2 Cloud, and building a monster server for under $3k.

Impact Linux, LLC http://impactlinux.com

Advantages of cross-compiling

- Why would you want to cross-compile?
- Fast
  - About as fast as native compiling on the host
  - Same SMP Scalability
- Prerequisites are common
  - Just software, common PC Hardware and Linux OS

Advantages of cross-compiling

- Unavoidable
  - Someone somewhere has to do a certain amount of cross-compiling in order to get a new target up and running
  - Minimal subset must be supported for each target
  - Can you build a current Linux 2.6 Kernel on an Alpha Machine running a 2.2 Kernel with GCC 2.95? If not, you need to cross-compile from a newer system.

Disadvantages of cross-compiling

- Keeping Multiple Build Contexts Straight
  - Information leaks between the target and the host
  - Each toolchain has its own headers and libraries
    - Different things in $PATH depending on target or host
    - Hard to remove things the host has, but the target doesn't
      - Install gzip on the host, but not in the target

Disadvantages of cross-compiling

- Keeping Multiple Build Contexts Straight
  - uname -m
    - Machine type given by the host may not equal target
  - Environment Variables
  - Looking at /proc

Disadvantages of cross-compiling

- Lying to Configure
  - The design of configure is fundamentally wrong for cross-compiling
  - Asking questions about the host to build programs for the target is crazy when the host and the target are not the same
  - libtool, pkg-config, finding Python, wrong signal numbers for Perl...

Disadvantages of cross-compiling

- Hard to test the result
  - The test suite is often part of the source. You didn't build the source on the target, so you have to copy the source tarball to the target hardware. Then to run its test suite you need to cross-compile tcl/expect and install it to the target's root filesystem.
- Do you want to ship that?
  - Will you bother?

Disadvantages of cross-compiling

- Bad Error Reporting
  - Using the wrong "strip" **mostly** works... until you build for sh4 target.
  - The Perl thing mentioned earlier shipped to a customer. It built, it ran, it just didn't work in this one area.

Disadvantages of cross-compiling

- Not all packages cross-compile
  - Most developers barely care about native compiling on non-x86, let alone cross-compiling
  - Implementing cross compiling complicates the heck out of the build system for everybody, to serve less than 10% of the userbase
  - Less than 10% of the userbase will ever test it, so it'll break a lot

Disadvantages of cross-compiling

- The package developer can't reproduce your bug.
- Developers haven't got a cross compiling build environment set up for every possible target. Installing a binary-only toolchain requiring a root login on their development machine isn’t appealing
- Developers haven't got a test environment to run the result. Building a binary they can't run proves nothing and isn't very interesting.

Disadvantages of cross-compiling

- The packages that do cross compile don't do it the same way, it's not just "./configure; make; make install".

Disadvantages of cross-compiling

- Maintenance
  - Each new release breaks
  - Discourages upgrading
    - MIPS was broken in the 2.6.30 kernel
    - sh4 is broken in 2.6.31
    - The kernel is **the** poster child for many eyeballs
    - Non-x86 gets much less attention than x86 at the best of times

Disadvantages of cross-compiling

- Now add 100 different build environments you could be cross compiling **from**. Add enough variables (which toolchain, which build scripts, which C library) and you may be the **only** person with your particular setup.
- Little or no regression testing means bit-rot.

Disadvantages of cross-compiling

- Open source community only cares if you're using current version
  - Debugging old versions doesn't help the project
  - Fixes only go upstream if applied to current
- Upshot
  - Most embedded projects get stuck using old packages, even when you **can** get all the source.

Disadvantages of cross-compiling

- Hairball Builds
  - everything has to know about everything else
  - buildroot wasn't initially intended to be a distro (embedded variant of Zawinski's law)
  - non-orthogonal, hard to mix and match toolchain, build scripts, packaging (rpm, dpkg, ipkg/opkg, portage, tarballs), system imaging
  - Build system generally picks one of each and requires you to use its choices together

Disadvantages of cross-compiling

- Hidden dependencies
  - The **host** has to be set up specially
    - This is never properly documented, and the docs bit-rot if they exist
  - It builds for this engineer but not that one
  - Try to reproduce 6 months later and something's changed

Disadvantages of cross-compiling

- Previous mention of installing a target version of gzip into your toolchain, not just into your target system, so later packages can build against it
  - Version in toolchain and target must match

Disadvantages of cross-compiling

- Pain to set up
  - Knowing how to build a working cross compiler toolchain yourself is a black art
  - Even using an existing toolchain build system ala crosstool-ng requires extensive configuration
  - Lots of toolchain build systems aren't standalone, using their toolchain for other purposes is an afterthought
  - Slightest mistake results in leaky toolchain

Disadvantages of cross-compiling

- Prebuilt binaries may or may not work for you unless you use the recommended version of the recommended distro
  - Linux has never been big into binary portability
- Lots of absolute paths hardwired into gcc. Can't just extract into your home directory, need root access to install it as a package

Disadvantages of cross-compiling

- Pain to set up
  - Most developers get a prebuilt BSP (Board Support Package, I.E. build system) and use that as a black box. (See hairball, previously)
  - Can't report bugs upstream because package developers won't set up 15 different BSPs. Which leads into...

Disadvantages of cross-compiling

- Hard to get package bugs fixed
- Can't report bugs upstream because package developers don't have your cross compiling environment set up and don't have the hardware to test the result anyway

Disadvantages of cross-compiling

- Restricted package selection
  - About 200-600 packages cross compile, depending on which platform you're targeting, how much effort you're willing to put in, and your definition of success
  - Debian has >30,000 packages. That's less than 2% of the total

Advantages of native compiling on real hardware

- Native compiling is simpler
  - No special support required in build
  - One context, no host/target split to leak information from
  - ./configure works as designed
  - Pushing bugfixes upstream takes less explanation

Advantages of native compiling on real hardware

- For something like an iPhone, doing this is almost feasible.
  - There are also high-end PPC and ARM workstations.
    - This is how Fedora for ARM is built, using a build cluster of high-end ARM systems.

Disadvantages of native compiling on real hardware

- Need to obtain and install a bootable development environment
- Chicken and egg problem
- Either cross compile from host or use prebuilt binaries

Disadvantages of native compiling on real hardware

- Is the device powerful enough?
  - Fast CPU, enough memory, enough disk, console access
    - 200 Mhz CPU with 32 Megs of RAM and a jffs2 flash root filesystem is not an ideal modern build environment
      - Flash has limited write cycles, building on it shortens device life.
    - Building on NFS is painful
      - Timestamp granularity and skew due to cacheing

Disadvantages of native compiling on real hardware

- Did you bring enough for everybody?
  - These systems can be expensive and hard to share.
  - Can you afford to buy one for each engineer?
  - Lots of money into rapidly depreciating non-general-purpose asset.
  - Is it really cheaper to manage resource contention when engineers try to share 'em?

Disadvantages of native compiling on real hardware

- Even when you can ssh into it and it's something high-end like an iPhone or a game console, does it have enough resources to run two builds at once?
- If not, sharing it between multiple developers is cumbersome

Disadvantages of native compiling on real hardware

- Headless box UI issues
  - Need lots of infrastructure before you can run "hello world"
    - Appropriately configured/packaged/installed: toolchain, hardware installer (JTAG?), bootloader, kernel, console (serial port? Network?), root filesystem, C library, init program
    - If any of that fails, you may not get **any** output. Good luck.

Disadvantages of native compiling on real hardware

- Easy to brick it
  - Need to reboot, re-flash, configure serial console. (See "sharing" above. If it's in the server room, you walk in there a lot or set up expensive infrastructure to automate power cycling, which has its own layer of control infrastructure.)
  - Can't necessarily just stick in a boot CD and start over

Disadvantages of native compiling on real hardware

- Reduces flexibility
  - You have to make your hardware decisions early in your development cycle, you can't do much software development until you have the hardware
  - You can always get it running first on x86 and then port it, which has its own issues
    - Since x86 is the default platform everything gets developed on, "it works on x86" doesn't prove much

Disadvantages of native compiling on real hardware

- Still tricky to send stuff upstream.
- Package maintainers are less likely to have your hardware than they are to have/install your BSP. They can't build/test your issues.

Disadvantages of native compiling on real hardware

- Less portable
  - A laptop can come with you on a plane, or work from home
  - Not all targets can be powered from USB or battery
- Not easily replaced
  - In case of a coffee spill you can get another PC and set it up same day

What do we want?

- It's possible to cope with the above, and a lot of people do. None of these problems are insurmountable, with enough engineering time and effort.
- But it's not ideal
- What have we identified so far that would bring us closer to an ideal build system?

What do we want?

- Scalable
  - Scale to large development teams without significant resource contention
  - Run on cheap personal machines like hobbyists have, so open source hobbyists can use/develop with it. (Minimally usable on a $250 Thinkpad.)

What do we want?

- Scalable
  - Reasonably fast
    - Take advantage of SMP (possibly even clustering for build servers) so you have the option to throw money at it to make it faster
  - Capable of building complicated projects without collapsing into a tangle or requiring days for each build cycle

What do we want?

- Reproducible
- Easy to modify and debug
- Breaking pieces you can't rebuild from source sucks
  - When something does break, you can make it happen again, take it apart and examine it, add logging, attach gdb to it, etc.

What do we want?

- Maintainable
- Something you can archive and retry years from now without having to dig up a copy of Red Hat 7.2 and try to get it to work on current hardware behind enough of a firewall it won't immediately get pwned by the ambient flora on the department's Exchange server.

What do we want?

- Simple
  - Understandable, easy to learn, debug, extend.
  - If it doesn't exist, it can't break.
    - Well, almost. Piggy's zero byte file story

What do we want?

- Orthogonal
  - Don't tie pieces together. Pick and choose.
    - Toolchain from here, build scripts from here, package management system from here...
    - Too many "accidental distros", like buildroot. Hairball is the natural state of cross compiling.

What do we want?

- Don't tie pieces together. Pick and choose.
  - If you wind up writing your own in-house (or forking an upstream BSP), maintenance snowballs. You'll wind up rewriting it from scratch every 5 years or so, without improving much.
  - Something you can re-use for your **next** embedded project.
  - Don’t let the tool take over your project

What do we want?

- Orthogonal
  - Not stuck to a single implementation.
  - Upgradeable
    - Drop in new upstream components as they release.

What do we want?

- Portable
  - Move build to new machines
  - Take build home, to coffee shop, on a bus, on a plane
  - What if your developer's laptop is a Mac or Windows machine?

What do we want?

- Self-contained
  - Something you can download/run without root access would be nice
    - Dear hobbyist developer, I found a bug in your code. To reproduce it, log into your laptop as root and go...
    - Easy enough to point upstream package managers at so they can reproduce your bug, diagnose the problem themselves, and merge a fix.

What do we want?

- Scriptable
  - Automated regression testing is nice.
  - Cron job, nightly build and test.
    - Including upstream component projects.

What do we want?

- Cheap
  - Free software running on cheap commodity x86-64 hardware.
  - Easy transition to your deployment environment (final hardware).

Compiling under emulation

- Native compiling under emulation can give us this.
- Moore's law is on your side.
  - It's already given us ridiculously fast SMP x86-64 hardware.

Compiling under emulation

- Others have better ratio of power consumption to performance, giving longer battery life, fanless operation, less weight, physically smaller, integrated peripherals...
- A few like the DEC Alpha, Sparc, and Itanic aimed for better absolute performance.
- They got steamrollered.

Compiling under emulation

- Often cheaper to throw hardware at the problem instead of engineering time. Moore's law does not make programmers 50% cheaper every 18 months.

Compiling under emulation

- QEMU is your friend.
- QEMU conceptually forked off of Fabrice Bellard's Tinycc project in 2003, originally as a way to run Wine on non-x86 hosts. In six years it's become a category killer in open source emulation projects.

Compiling under emulation

- QEMU is your friend
  - QEMU does "dynamic recompilation" on a per-page basis. As each page of executable code is faulted in, QEMU translates it to an equivalent chunk of host code. It keeps around a cache (~16 megabytes) of translated pages, so that running the same code multiple times (loops, function calls) doesn't re-incur the translation overhead.

Compiling under emulation

- The reasons this is slower than native performance are:
  - Translation overhead
  - (essentially increased page fault latency)
  - The translated code is less efficient because any serious optimizer would slow down the translation too much

Compiling under emulation

- That said, QEMU's ARM emulation on a 2Ghz x86_64 is likely to be faster than a real 400mhz ARM920T.
  - Good rule of thumb is 20% of native speed, but it varies target, per QEMU version, per load...
    - Bench it yourself.

Compiling under emulation

- There are several other open source emulation projects but:
  - They don't emulate different targets like qemu does, just host-on-host (generally x86 on x86).
  - Most of them use QEMU code to emulate I/O devices because QEMU has the best device support (not counting 3D acceleration hardware).

Compiling under emulation

- Mostly survive in special purpose niches:
  - Virtualbox (which was sold to Sun which was sold to Oracle) and Xen (which was sold to Citrix) are kept alive by for-profit corporate owners, and are mostly targeting non-Linux hosts since KVM was (recently) integrated into QEMU and the Linux kernel.

Compiling under emulation

- Valgrind is really a debugger
- Bochs is largely an x86 bios project these days
- User Mode Linux and Iguest are the Linux kernel emulating itself.

Compiling under emulation

- There are other non-x86 emulators (aranym for m68k, Hercules for S/390), and the concepts here apply to them too, but QEMU will probably eat them eventually.
- They generally focus on a single target. QEMU emulates many targets, so you can apply what you've learned to future projects.

Compiling under emulation

- QEMU has two modes:
  - Application and System Emulation

QEMU Application Emulation

- This mode runs a userspace program, intercepting system calls and translating them to/from the host OS.
- Good for "smoke testing" your toolchain and C library: run a statically linked "hello world" program to make sure your toolchain (compiler, linker, C library, kernel headers) is set up correctly for your target.

QEMU Application Emulation

- Some projects (such as some versions of scratchbox) set up the kernel's "misc binary support" to call QEMU application emulation to run target binaries.
  - This is to placate ./configure, and builds that aren't cross compile aware.

QEMU Application Emulation

- This is only a partial fix, leaving many problems unaddressed. The resulting build is still fairly brittle, very complicated, and requires root access on the host to set up, but it's a viable approach.
  - Also needs a version of uname that lies about the host, and some fiddling with shared library paths to run non-static target binaries, and remember to export HOSTTYPE and MACHTYPE environment variables...

QEMU Application Emulation

- QEMU Application Emulation is nice but limited

QEMU System Emulation

- Standalone build system isolated from the host.
  - The rest of this talk describes this approach.
  - Advantages
    - See "What We Want" previously. All that applies here

QEMU System Emulation

- Disadvantages
  - As with native hardware build, need to find/create a development system
  - Install's a lot easier though
  - Emulation overhead, slower than native compiling
  - Needs extra work to take advantage of SMP

QEMU System Emulation

- Disadvantages
  - Sometimes building on native hardware wins (faster, more convenient, more accurate, etc.)
  - There exists hardware with no good emulation
  - Running on the emulator doesn't always prove it'll work properly on the real hardware
  - You can always do both. (See "orthogonal" above, even QEMU can be swapped out.)

Obtaining a development environment for QEMU

- Root filesystem and native toolchain for target hardware, plus kernel configured specifically for one of QEMU's system emulations.
- First question: Use prebuilt binaries or build from source?

Using Prebuilt Binaries

- Prebuilt binaries are easier but less flexible.
  - They save time up-front at the risk of costing you a **lot** of time if anything goes wrong
  - If your intention is to put together a new embedded system, putting together your own development environment is good practice
  - Tailoring a system to save space/power/latency means eliminating unnecessary packages/libraries/daemons

Using Prebuilt Binaries

- There are lots of prebuilt distros to choose from.
  - Your hardware vendor probably has a BSP you can repackage for QEMU.
  - They may not offer a native toolchain, though.
  - Angstrom Linux, Gentoo Embedded, Emdebian, Openwrt, Openmoko, Fedora for ARM, Buildroot, and many many more.

Using Prebuilt Binaries

- No guarantee any of them will meet your needs out of the box.
  - May need to repackage them for QEMU if they don't explicitly target it.
  - New kernel, drivers
  - ext3 filesystem image (with native toolchain and prereq packages)
  - QEMU command line.

Using Prebuilt Binaries

- Even if you're not personally going to build a development environment from source, it's helpful to understand how one is put together.
- Learning how it works can only help
  - Unless you're using Windows, which stops working when you collapse its quantum state.

Building a development environment from source

- Using a build system without understanding how it works isn't that much better than just grabbing prebuilt binaries.
- But it does give you a fall back position if something goes wrong, you can then start learning about the build system when you need to.
- Thinking ahead, you might want to pick a build system that isn't too hard to learn if you **do** need to take it apart

Building a development environment from source

- The simpler your build system is, the less there is to go wrong
  - Less to keep up-to-date
  - Easier to learn
  - If cross compiling is inherently harder than native compiling, then the most maintainable system is probably one that does as little cross compiling as possible before switching over to a native build

Building a development environment from source

- This is similar to the way tailoring an embedded system to save space/power/latency means eliminating unnecessary packages/libraries/daemons.
  - Less is more
- That said, the less the build system starts with, the more you'll have to add to it to meet your needs. Dropbear, strace, gdb, gzip....

Understanding your build environment

- The best educational reference for this area is Linux From Scratch at http:/www.linuxfromscratch.com
  - Excellent educational resource. It is explicitly **not** a build system. (Not automated, you cut and paste chunks to run them as you read.)
  - It describes how to create a fairly big system by embedded standards, around 100 megabytes.

Understanding your build environment

- Linux from Scratch
  - Uses "standard" GNU bloatware. No busybox, uClibc, dropbear, full System V boot scripts
  - Conservative approach, builds prerequisites for test suites (such as tcl/expect) as part of base system and runs each test suite during build.

Understanding your build environment

- Linux from Scratch
  - Based on native compiling, building a new system of the same type as the host.
  - There's a Cross Linux From Scratch that covers cross compiling, but it's much newer, less thorough, less easily understandable, still a bit buggy in places, and not very actively developed (Last release was three years ago).
  - Read (and understand) the original one first

Reinventing the wheel

- Many warnings (Danger Will Robinson)
  - On reinventing the wheel, and the hairball problem
  - Just about every new distro or build system in the past decade has started with somebody writing a shell script to automate the Linux From Scratch steps, and then tweaking it. You are not the first, and most of the others have a 10 year headstart on you.

Reinventing the wheel

- If you do want to create your own embedded build system anyway, expect it to take about a year to get something reasonably usable. (It's a great learning experience.)
- Maintaining your own build system is enormously time consuming.

Package Management and Accidental Distros

- Linux From Scratch intentionally does not cover package management.
- Package management two parts: Using a package manager (rpm, dpkg, ipkg/opkg, portage) to install/uninstall/track packages, and creating a repository of build scripts and package dependencies.

Package Management and Accidental Distros

- Creating a repository of package dependencies and build options is a huge undertaking and requires constant maintenance as new packages are released upstream.
- Fedora, Debian, Gentoo, and others already have perfectly acceptable repositories you can leverage. Creating your own is a huge time sink.

Package Management and Accidental Distros

- Despite this, it's very easy to fall into the trap of creating your own Linux distro by mistake, at which point complexity explodes out of control.
- Maintaining your own distro is enormously time consuming.
- The more packages your build system builds, the more likely this is to happen.

Package Management and Accidental Distros

- Defending yourself from accidental distros.
  - Divide the hairball into orthogonal layers. Toolchain selection has nothing to do with package management
  - Needing to change one is no excuse for accepting responsibility to maintain the other.
  - Delegate everything you can to existing projects.
    - Push your changes upstream, which makes them somebody else's problem.

Package Management and Accidental Distros

- Figure out what your goals are and what you're **not** going to do
  - You can't stay focused if you can't say no

Buildroot Example

- The **buildroot** project was originally just a uClibc toolchain creator and test harness. The easy way to test that a package built against uClibc was to add it to buildroot, and since it never had a clear design boundary allowing it to say "no" to new features, this quickly grew out of hand.

Buildroot Example

- The project maintainer (Erik Andersen) and several of his senior developers had so much of their time taken up by buildroot they stopped contributing to the original project, uClibc.

Buildroot Example

- The uClibc mailing list was swamped by buildroot traffic for years until Rob created a new buildroot list and kicked the traffic over there, but uClibc development still hasn't fully recovered.
- This is a significant contributing factor to uClibc's TLS/NPTL support being feature complete for its first architecture in 2006 (there was an OLS paper on it) but still not having shown up in an actual uClibc release 3 years later.

Buildroot Example

- They do the distro thing badly: no package manager, no repository.
- Buildroot didn't use a package manager (like rpm, dpkg, ipkg/opkg, or portage), instead it encoded its dependency resolution in menuconfig's kconfig files, and the package build logic in a series of nested makefiles, then stored the whole thing in source control. Thus the builtroot develpers took a while to notice it was becoming a distro.

Buildroot Example

- Lots of open source build systems forked off of buildroot, just as lots of in-house build systems forked off of board support packages. They quickly diverge enough to become independent projects capable of absorbing enormous amounts of time.

Buildroot Example

- Moral: Original project stalled because its developers sucked away, wound up doing the distro thing badly with no package manager, no repository, no stable releases.
- Don't let this happen to you.

Orthogonal layers

- The following is based on what our Firmware Linux project does.
  - Other build systems need to do the same things, most of them just aren't as careful to separate the layers.

Orthogonal layers

- Each of these layers can be swapped out
  - Use somebody else's cross compiler, use our cross compiler to build somebody else's root filesystem, package an arbitrary directory with our packaging script, etc.
  - We try to make it as easy as possible to NOT use our stuff, and show it here only as an example.

Orthogonal layers

- This build is intentionally implemented as a simple series of bash scripts, so you can read the scripts to see what it's doing.

Downloading Source

- Download source code (download.sh)
  - Calls wget on a bunch of URLs to copy files into "packages" directory.
  - You can just cp 'em there yourself instead if you like

Downloading Source

- Does security/integrity sha1sum checks on existing tarballs (if any)
  - Some build systems (such as crosstool-ng) give the weirdest errors if you re-run the build after an interrupted download, and you have to figure out the problem and fix it by hand
  - The GNU ftp server has been cracked before

Downloading Source

- Cache tarballs so once you've run this step you no longer need net access
  - Falls back to a couple mirrors if the source site is down
  - PREFERRED_MIRROR environment variable points to a mirror to try first if you have a local copy. Note that QEMU's 10.0.2.2 tunnels through to the host's 127.0.0.1, so if your build host runs a web server on loopback it can easily provide source tarballs to a build inside QEMU without bogging the public servers.

Downloading Source

- Track all the source URLs and versions in one place
- Only this file has version info in it, making drop-in upgrades a two-line change.
  - Non-drop-in upgrades require debugging, of course

Host Tools

- Entirely optional step, can be skipped if your host is set up right, but that's seldom a good idea.
- Serves same purpose as LFS chapter 5: an airlock preventing leakage.
  - LFS chroots, FWL adjusts the $PATH to remove stuff.
  - Remember, adding stuff is easy. Removing stuff the host already has so cross compiling doesn't accidentally find and leak it into your build is one of the hard parts.

Host Tools

- Provides prerequisites automatically, no need to install packages by hand as root
  - Nothing the FWL build system does requires root access on the host. That's an explicit design goal.
- Isolate your build from variations in the host distro
- Provide known versions of each tool

Host Tools

- Smoke test and regression test
  - Building everything from here on with the same tools we're going to in the final system (busybox, etc) demonstrates that the build system we're putting together can reproduce itself.
  - And if it can't (regression) we find out early

Create a Cross-Compiler

- Four packages: binutils, gcc, linux kernel headers, uclibc
  - In that order
    - gcc depends on binutils, building uClibc depends on all three previous packages.
  - Later, we add a fifth package: uClibc++, for c++ support.

Create a Cross-Compiler

- Use a compiler wrapper (ccrap.c) based on the old uClibc wrapper to make it all work together
- The wrapper parses the gcc command line and then rewrites it to start with --nostdinc and --nostdlib, then explicitly adds in every library and header path gcc needs, pointing to a known location determined relative to where the wrapper binary currently is.

Create a Cross-Compiler

- This prevents the host toolchain's files (in /usr/include and /lib and such) from leaking into the target toolchain's builds.
- It also means the toolchain is "relocatable", I.E. it can be extracted into any directory and run from there, so it can live in a user's home directory and does not require root access to install.
- Just add the toolchain's "bin" subdirectory to your $PATH and use the appropriately prefixed $ARCH-gcc name to build.

Create a Cross-Compiler

- The simple compiler above can be created without prerequisites
- Thus creating a simple cross compiler is always the first step

Create a Cross-Compiler

- For FWL releases we build a second cross compiler that's statically linked against uClibc on the host, which is a two-stage process
  - First build an i686-gcc compiler, which outputs 32-bit x86 binaries
  - Then rebuild the cross compilers as --static binaries using that as the host compiler

Create a Cross-Compiler

- This increases portability and decreases size. You can run the resulting binaries on 32 bit Red Hat 9 and on 64-bit Ubuntu 9.04.
- This one also builds uClibc++, an embedded replacement for libstdc++
  - The first toolchain can build C++, but has no standard C++ library to link them against

Create a Cross-Compiler

- --disable-shared vs --enable-shared
- History of ccwrap
  - uClibc compiler wrapper
- evolution of buildroot
- Why building libstdc++ is painful

Cross-compile a root filesystem

- Long story short: seven packages (busybox, uClibc, linux, binutils, gcc, make, bash), plus some empty directories and misc files like an init script.
- This creates a directory full of files, suitable to chroot into.

Cross-compile a root filesystem

- Create directory layout
  - tmp proc sys dev home usr
  - Symlinks into usr/{bin,sbin,lib,etc} from top level.
    - Historical note where "usr" came from in 1971.

Cross-compile a root filesystem

- Install uClibc
  - Build this first so everything else can build against it.
  - Cross compiler may not come with libc we're installing, so use ccwrap on existing cross compiler here.
    - This is redundant when using **our** compiler, but necessary for orthogonality.
      - Allows you to use arbitrary existing cross compiler

Cross-compile a root filesystem

- Build busybox
  - Provides almost all POSIX/SUSv4 command line utilities
  - Switch off a few commands that don't build on non-x86
  - Add a few supplemental commands (getent, patch, netcat, oneit) from "toybox" or as shell scripts. Mostly optional.
    - Biggest issue is that busybox patch can't apply hunks at an offset, which makes it too brittle to use in a development environment

Cross-compile a root filesystem

- Install native toolchain
  - Same packages as cross compiler, plus uClibc++
  - Note, FWL builds a statically linked native compiler, which you can extract and run on an arbitrary target system.
  - Same theory as the statically linked cross compiler, only for native builds on systems that don't come with a uClibc toolchain.

Cross-compile a root filesystem

- Install make, bash, and (optionally) distcc.
  - You can't do ./configure; make; make install without "make". (While cross compiling, used the host's version of make.)
  - Build bash because busybox ash isn't quite good enough yet
    - Getting pretty close, need to retest and push fixes upstream
    - Building bash 2 not bash 3, because it's 1/3 the size, and doesn’t require ncurses

Cross-compile a root filesystem

- Install init script and a few misc files
  - Copy sources/native/* to destination
  - Includes an init script
  - An etc/resolf.conf for QEMU
  - Some "hello world" C source programs in usr/src...

Cross-compile a root filesystem

- The init script does the following
  - Mount sysfs on /sys, and tmpfs on /dev and /tmp.
    - tmpfs is a swap-backed ramfs, writeable even if root filesystem is read-only.
  - populate /dev from /sys (using busybox mdev instead of udev)
  - Configure eth0 with default values for QEMU's built-in vpn.

Cross-compile a root filesystem

- The init script does the following
  - Set clock with rdate if unix time < 1000 seconds.
    - This means this emulation has no battery backed up clock.
    - Uses host's inetd on 127.0.0.1 to get current time.
  - If we have /dev/hdb or /dev/sdb, mount it on /home.

Cross-compile a root filesystem

- The init script does the following
  - It echos "Type exit when done.", which means the boot was successful and gives "expect" something to respond to.
  - Use "oneit" program to run a command shell.
    - This gives us a shell with a controlling tty (ctrl-c works), and the system shuts down cleanly when that process exits.

Cross-compile a root filesystem

- Optimizations
  - Strip binaries, delete info and man pages, and gcc's "install-tools".
  - Build without a native toolchain
    - Means you can also delete /usr/include and /usr/lib/*.a from uClibc.
    - Don't need make or bash

Cross-compile a root filesystem

- Optimizations
  - Build static or dynamic
    - When building static and not installing a native toolchain, we can skip this step entirely. Just use whichever libc the cross compiler has, and don't bother to install it on the target.

Package a system image

- qemu-system-* needs a bootable kernel and a filesystem image.
  - So does real hardware.
- Build a bootable Linux kernel
  - Needs correct CPU and board layout (device tree or equivalent), drivers for serial console, hard drive, network card, clock, ext2/ext3/squashfs/tmpfs

Package a system image

- Which kernel .config do you build?
  - The kernel has default configuration files for a lot of boards.
  - Look in the Linux kernel source for an arch/$ARCH/configs directory
  - Or a single arch/$ARCH/defconfig file for the less-diverse architectures

Package a system image

- We use the "miniconfig" technique
  - Uses miniconfig.sh to create one
  - make allnoconfig KCONFIG_ALLCONFIG=filename
  - Portability, Readability
  - Also works with uClibc and busybox .configs.

Package a system image

- Use mksquashfs, gene2fs, or cpio/gzip to create standalone filesystem image.
  - Possibly rebuild kernel to include initramfs (gzipped cpio image).
    - Note initramfs is size-limited by most kernels' memory mappings, and seldom has enough space for a dozen megabytes of toolchain.

Alternatives to packaging

- All sadly inferior, but know your options
- QEMU Application Emulation
  - It can run programs out of root-filesystem.
    - Requires root access to use the -chroot option (and to populate /dev).
    - Not much faster than full system emulation, but significantly more brittle.

Alternatives to packaging

- Strangely, application emulation is a harder problem to solve than system emulation.
  - System emulation starts from scratch and fakes something simple, a CPU, some attached memory (with MMU), and a half-dozen well-documented device chips (clock, serial, hard drive, etc).

Alternatives to packaging

- QEMU application emulation must intercept literally thousands of syscalls and ioctl structures (each of which may have a dozen members) and translate every one its own special case, dealing with endianness and alignment and sometimes different meanings of the fields on different architectures.

Alternatives to packaging

- Signal numbers differ.
- Page sizes differ, which affects mmap behavior in several ways (padding the end of the file with zeroes for **how** long?)
- It's **fiddly**.

Alternatives to packaging

- QEMU application emulation is a nice smoke test, but you wouldn't want to develop there.
  - Run a static "hello world" to see if your compiler's working.
  - Fewer prerequisites between you and a simple shell prompt.
    - No need to build a kernel, let alone get serial console working.

Alternatives to packaging

- QEMU can create virtual FAT disk images
  - Option "-hda fat:/path”
    - This exports a directory as a virtual block device containing a read-only vfat filesystem constructed from the files at path.
    - Remember to mount it read-only: if Linux tries to write to it lost interrupts and driver unhappiness ensue.

Alternatives to packaging

- Standard "booting from vfat" issues
  - Case insensitivity, no ownership, incomplete permissions...
  - Doable, but fiddly.
- Do **not** change contents of host directory while QEMU is running, it'll get very confused.

Alternatives to packaging

- Assembling a virtual FAT with lots of little files can take lots of time and memory
  - Doesn't scale. QEMU may grind for quite a while before launch with even a relatively small filesystem.

Alternatives to packaging

- Exporting a directory via a network filesystem is another way to provide a root filesystem to QEMU or to real hardware.
  - May be OK for read-only root filesystem, not so good for writeable development space you compile stuff in.
  - Login management unpleasant, make sure you do this behind a firewall or via loopback (10.0.2.2 in qemu is 127.0.0.1 on host).

Alternatives to packaging

- Linux's most commonly used network filesystem (NFS, Samba, TFTP) all **suck** in various ways.
  - NFS sucks rocks
    - The words "stateless" and "filesystem" do not belong in the same sentence. (Thanks Sun!)
    - Requires root access on the host to export anything
    - Fiddly and brittle to automate, so usually done by hand

Alternatives to packaging

- Building on non-readonly NFS is unreliable due to dentry cacheing issues
  - Make gets very confused when timestamps go backwards.
  - General problem: either you have network round trip latency for each directory lookup or you have local dentry caches that can get out of sync with the server's timestamp info.
  - At least most things don't care about atime.

Alternatives to packaging

- Samba sucks almost as many rocks
  - It's a Windows protocol, constructed entirely out of weird non-POSIX behavior and corner cases.
  - Case insensitive, many builds don't like this.
  - Dentry info translated (can confuse make), doing things like mmap (and even symlinks) hacked on.
  - Share naming and account management are their own little world.

Alternatives to packaging

- Finding the server can be a pain
  - Domain server and Active Directory and such: there be dragons
  - But at least it's got a userspace server that can export a filesystem running as a normal user, and a lot of people already understand it quite well.
  - QEMU has a built in -smb option, which launches samba on the host for you to export the appropriate directory, visible inside the emulator on 10.0.2.4 as "\\smbserver \qemu".

Alternatives to packaging

- TFTP
  - Only mentioned because QEMU has a built in -tftp option, which exports a directory of files to the emulator via a built-in (emulated) tftp server.

Alternatives to packaging

- Filesystem in Userspace (sshfs and friends)
- Can't boot from FUSE, need another root filesystem (generally initramfs) for setup, so doesn't solve this problem.

Booting a system under QEMU

- The command is "qemu-system-$ARCH"
  - Use tab completion to see the list
  - "qemu" should be called "qemu-system-i386" and have a symlink from "qemu" to your host.
  - Most hosts these days are **not** 32 bit, so "qemu" on them should probably point to qemu-system-x86_64.

Booting a system under QEMU

- Not only many different binaries, but also sub-options within each.
  - Try "-M ?" to list board types and "-cpu ?" to list processor variants.
  - Which boards have what hardware is at http://www.qemu.org/qemu-doc.html (mostly in chapter 4).
  - The -cpu option affects choice of cross compiler and root filesystem.
  - The -M options just affects kernel .config.

Booting a system under QEMU

- qemu-system-arm handles both endiannesses, but qemu-system-mips/mipsel are separate. qemu-system-x86_64 has all the 32-bit -cpu options, but there's a 32 bit qemu also.

Booting a system under QEMU

- Running a binary on a Pentium doesn't prove it'll run on a 486.
  - Unaligned access works just fine on an x86 host, but should throw an exception on things like m68k or ARM. QEMU may allow it anyway.
  - Newer versions of QEMU are a lot better about catching that sort of thing (they're implementing the fiddliest weird corner cases these days), but "runs in the emulator" never proves it'll run on the real hardware.

Booting a system under QEMU

- Emulation is great for development and initial testing, but there's never any real substitute for testing in situ.
- Your board may have peripherals QEMU doesn't emulate anyway.
  - Note you can tunnel any USB device through to QEMU, check the QEMU documentation for details.

Booting a system under QEMU

- Sometimes you want a pickier -cpu setting than QEMU currently provides
  - ppc440 code can often run on full ppc (it's mostly a subset of the instruction set), but not always, and not the other way around. But although qemu-system-ppc has an "-M bamboo" board, it doesn't have ppc440 -cpu option.
  - My first attempt at and "armv4lt-eabi" target worked on qemu's default CPU (an armv5l) but didn't work on real arm920t hardware.

Booting a system under QEMU

- QEMU development is rapid, and they're adding more and more granularity all the time. Check back, ask on the list, write it yourself, offer to sponsor the existing developers to work on your feature (several of them work for codesourcery.com)...

QEMU Command Line Options

- Command line options to qemu-system-$ARCH:
  - Use "-nographic" to make qemu scriptable from the host.
    - Disables video card emulation, and instead connects the first emulated serial port to the QEMU process's stdin/stdout.
    - This gives you an emulator shell prompt about like ssh would, in a normal terminal window where cut and paste work properly.

QEMU Command Line Options

- You might want to build and use "screen" inside the emulator, or set $TERM, $ROWS and $COLUMNS for things like busybox "vi".
- This means you can run QEMU as a child process and script it with "expect" (or similar).
  - This is marvelous for automating builds, regression testing, etc.
  - Wrap a GUI (or emacs) around the QEMU invocation.
    - QEMU runs on non-Linux hosts, no reason your GUI couldn't run on a Mac

QEMU Command Line Options

- Note, some versions of QEMU are built with a broken curses support that craps ANSI escape sequences randomly to stdout, which will confuse expect. If you encounter one of these, rebuild it from source with --disable-curses.
  - In theory this shouldn't happen if stdin is not a tty, but there are buggy versions out there that don't check.

QEMU Command Line Options

- You can also cat a quick and dirty shell script into QEMU’s stdin for the emulated Linux's shell prompt to execute.
  - The reason this is "quick and dirty" is it isn't 100% reliable, because the input is presented to stdin before the kernel has a chance to boot.
  - Serial port initialization eats a variable amount of data, generally around the UART buffer size (16 bytes-ish for a 16550a, but there's some race conditions in there so it's not quite contstant).

QEMU Command Line Options

- This is a Linux kernel serial/tty issue, not a QEMU issue.
- You can work around this by starting your script with a line of space characters and a newline, and then it'll work ~99% of the time. But if you use it a lot (automated regression, nightly builds, etc) the lack of flow control can inject the occasional spurious failure because the script didn't read right.

QEMU Command Line Options

- Using "expect" (or similar) is more reliable. The kernel doesn't discard data when it's **listening** for data. The tty driver and various programs just get a bit careless with the buffer if you get too far ahead when they're **not** listening for data.

QEMU Command Line Options

- Use "-no-reboot" so QEMU exits when told to shut down by the kernel.
  - Helps with scriptability.

QEMU Command Line Options

- Use "-kernel FILENAME" to specify which kernel image to boot.
  - Avoids the need for grub or u-boot to launch emulated system.
  - Uses a very simple built-in bootloader to load kernel image into QEMU's emulated physical memory and start it.

QEMU Command Line Options

- QEMU has code to boot an ELF format linux kernel
  - vmlinux in top level kernel build
    - Alas, not enabled for all architectures
  - Can also handle bzImage and various target-specific variants

QEMU Command Line Options

- "-hda FILENAME"
  - Provides virtual ATA/SCSI block devices for board emulation
    - Specify more with -hda, -hdb, -hdc...
      - First one can be a read-only root filesystem, second writeable scratch space to compile stuff in, third extra data such as source tarballs...
      - Yes the option is "-hda" even when the board is emulating SCSI and thus the device it emulates becomes "/dev/sda".
      - No need to partition these, Linux can mount /dev/hda just fine

QEMU Command Line Options

- "-append 'OPTIONS'"
  - The Linux kernel command line
    - In theory this is extra stuff added to the default kernel command line, but in practice it's generally the whole kernel command line.

QEMU Command Line Options

- Relevant options you probably want to specify:
  - "console=ttyS0" (or platform equivalent device)
    - Tell the kernel to send boot messages (and /dev/console, thus the stdin/stdout of PID 1) to a serial device
    - I.E. "serial console"

QEMU Command Line Options

- "root=/dev/hda"
  - Specify where the root filesystem lives
  - Also specify "rw" so root filesystem is read/write
  - Don't need either if booting from initramfs

QEMU Command Line Options

- "init=/usr/sbin/init.sh"
  - Specify what executable file to launch as the initial process. The "init task" (PID 1) is special. It has signals blocked, the kernel panics if it ever exits, it gets sigexit for orphanedchild processes and has to reap their zombies, etc
  - Note that just using init=/bin/sh will give you a quick and dirty shell prompt, but you won't have a controlling tty, thus ctrl-c won't work.

QEMU Command Line Options

- Meaning you won't be able to break out of simple things like "ping". Very easy to lose control, gets old fast.
- Also need to set up /proc yourself, populate /dev, ifconfig...

QEMU Command Line Options

- "panic=1"
  - This tells the kernel to reboot one second after panicking
    - Which -no-reboot above turns into QEMU exiting

QEMU Command Line Options

- Put it all together and you have something like:
  - **qemu -nographic -no-reboot -kernel zImage-i686 -hda image-i686.sqf -append "root=/dev/hda rw init=/usr/sbin/init.sh panic=1 PATH=/usr/bin console=ttyS0"**

Troubleshooting

- Do you see kernel boot messages on QEMU's stdout?
  If not:
  - Your serial console might not work
    - Linux serial driver
      - static driver, not a module. * instead of m.
      - console= line on kernel command line
      - Is QEMU's board emulation giving you the serial device you expect?

Troubleshooting

- Is my toolchain even producing the right output?
  - Build a statically linked "hello world" and run that on the host using QEMU's application emulation.
    - If it won't run, use the "file" command on the binary (and the .o and .a files) and check that it the kind of binary you think it is.
    - If the toolchain you're using uses ccwrap, try setting the environment variable WRAPPER_DEBUG=1 to see all the component files linked into the binary, and check them with "file".

Troubleshooting

- Kernel may not be booting up far enough to spit anything out
  - Is it the right configuration?
    - Enable EARLY_PRINTK in the kernel
  - Is it packaged the right way?
    - vmlinux, bzImage, chrp...

Troubleshooting

- Do you see boot messages but no shell prompt?
  - Ok, where does it stop?
  - Seeing some output is always better than seeing no output
  - You can always stick more printk() calls into the kernel source code

Troubleshooting

- Is it hanging on a driver before it tries to launch init?
  - Make sure the board emulation and kernel .config agree.
  - If it's a target that needs a "device tree", what does that say?

Troubleshooting

- Not finding the root filesystem?
  - Is the root= line correct?
  - Correct block driver?
  - Correct filesystem format driver?
  - Make sure those two drivers are static, not modules.
  - Try an initramfs?

Troubleshooting

- Mounting but unable to run init?
  - Check your filesystem image (via loopback mount) to make sure the binary you expect really is there.
  - Note that an initramfs uses "rdinit=" instead of "init=".

Troubleshooting

- Try pointing init= at a statically linked "hello world" program.
  - FWL contains one at /bin/hello-static
  - If this works, your dynamic linker is probably at fault.

Troubleshooting

- Then try a dynamically linked hello world. If that fails:
  - Is your library loader (/lib/ld-uClibc.so.0 or similar) where you think it is?
    - ldd is target specific but "readelf" is fairly generic.
    - Make sure all the libraries it tries to link to are there.
      - Remember: Shared libraries link to other shared libraries
      - run ldd/readelf on all

Troubleshooting

- Run "file" on everything to make sure no host binaries have leaked into your root filesystem.
- Maybe dynamic loading isn't supported with that libc on that target? (*cough* uClibc on Sparc *cough*)

Troubleshooting

- If it launches init but you get no further output.
- Is your init trying to redirect /dev/console and missing the serial console?
  - Work your way up from init=/helloworld through a shell prompt to whatever your init program is doing.
    - Add our prebuilt static strace binaries to your root filesystem and run your init under that.

Troubleshooting

- strace turns random hangs into "it made it **this** far, and was trying to do **that**"
  - Only useful if you're getting output from "hello world".

Troubleshooting

- Target-specific issues
  - ARM, MIPS, PowerPC, SPARC, sh4, x86, x86_64, M68k...

Using your emulated development environment

- The emulated development environment gives you two things
  - A scriptable build environment, so you can run nightly automated regression testing cron jobs to build and test your software
  - Not a substitute for testing on real hardware, but a good automated smoke test
  - A shell prompt at which you can do fresh development

Using your emulated development environment

- As with all new development environments, you may need to build/install lots of prerequisite packages to get the development environment you really want
  - gzip, ncurses, openssl, strace, dropbear, screen...

Using your emulated development environment

- Remember: what your development machine needs and what your deployment environment needs aren't necessarily the same thing.
  - Your development image has a toolchain. Probably won't ship that.
  - Quite possibly other stuff you'll develop with but not ship.
    - Needing to build it != needing to ship it.

Using your emulated development environment

- Editing source on your host system is a lot more comfortable.
  - Trying to add too much UI to the emulators is probably a waste of time. Your host is pretty and user friendly already
  - busybox has a "vi" implementation, so if you really need to you can edit source inside the emulator.
  - Using "screen" (which requires ncurses) makes things much less painful.
    - Probably need to export $TERM, $LINES, and $COLUMNS by hand to get it to work, ncurses can't query a serial console for tty info

Using your emulated development environment

- You probably still want to do most of your editing and source control and such on your host, in an IDE, with multiple xterms and source control and so on.
- So the problem becomes getting source from your host into your build system (and updating it), and getting results back out.

Getting data in/out of the emulator

- Your three main options are:
  - stdin/stdout
  - We mentioned the cat/expect scriptability.
  - Also cut and paste.
  - Good for small stuff, but doesn't scale to bulk data.

Getting data in/out of the emulator

- Filesystem images
  - Previously discussed
    - Awkward, but sometimes good for really large amounts of data.
    - Loopback mounting them on the host requires root access.
      - Can get data in without root access, harder to get it out.
      - genext2fs doesn't come with a corresponding extract

Getting data in/out of the emulator

- Virtual Network
  - The most powerful/flexible mechanism. This is the one to focus on.

Getting data in/out of the emulator

- Understanding the QEMU "user" network.
  - The QEMU docs at http://www.qemu.org/qemu-doc.html describe this.

Getting data in/out of the emulator

- QEMU defaults to providing a virtual LAN behind a virtual masquerading gateway, using the 10.0.2.x address range.
  - If you supply no -net options to QEMU, it defaults to "-net nic,model=e1000 -net user"
    - The first option plugs a virtual ethernet card into the virtual PCI bus, which becomes eth0.
      - Used to be ne2k_pci, switched to e1000 in 0.11.0 release.
      - The second option connects the most recent network card to the virtual LAN.

Getting data in/out of the emulator

- The virtual LAN provides:
  - A virtual masquerading gateway at 10.0.2.1. Works like a standard home router (ala Linksys, D-Link, etc): you can dial out but nobody can dial in.
  - A tunnel from 10.0.2.2 to the host's 127.0.0.1 (loopback interface)
    - An emulated Linux system has its own loopback interface, so if you want to connect to services running on the host's loopback you need to use this alias for it.
    - This comes in handy all the time.

Getting data in/out of the emulator

- A virtual DNS server on 10.0.2.3.
  - Receives DNS requests and resolves them using the host's libresolv.so and /etc/resolv.conf.
    - Any cacheing is the host's responsibility

Getting data in/out of the emulator

- A virtual DHCP server.
  - This responds to any DHCP requests by assigning it the address 10.0.2.15, with the above virtual gateway and DNS server addresses.
  - Since these are constant values, you might as well just assign them statically with ifconfig, route, and /etc/resolv.conf.

Getting data in/out of the emulator

- This is really useful, it means the emulated target system has a net connection by default.
- And it can use 127.0.0.1 to dial out to the host and talk to private local network serves, such as ssh, web servers, samba...
- But it's masqueraded. The host can't dial in.

Getting data in/out of the emulator

- The easy way to let the host dial in is port forwarding.
  - Again, just like a home router.
  - Use the "-redir" option
    - **qemu -redir tcp:127.0.0.1:9876::1234**
    - This means any attempt to connect to port 9876 on the host's loopback interface will be forwarded to port 1234 on the target's 10.0.2.15.
    - The :: is intentional, you could put 10.0.2.15 there if you wanted to, but that's the default value.

Getting data in/out of the emulator

- Using the virtual network to get data in/out of QEMU
  - busybox contains wget
    - Helps to run a web server on the host's loopback interface.
    - busybox also contains one of those, no reason you can't build it onthe host. If you run it on a port > 1024, it doesn't require root access.
    - There's always Apache, if you're up for configuring it
    - A public web server or department server works too

Getting data in/out of the emulator

- The netcat/tarball trick
- Build host and target versions of a netcat with a "server" mode (such as the one in toybox), and you can do tricks like the following shell script snippet:

Getting data in/out of the emulator

```
# Remember the first line of couple dozen spaces; Linux kernel serial
# port initialization eats about a FIFO buffer's worth of input.

./run-qemu.sh << EOF

[ -z "$CPUS" ] && CPUS=1

rm -rf /home/temp
mkdir -p /home/temp
cd /home/temp || exit 1

# Copy source code from host to target system
netcat 10.0.2.2 $(netcat -s 127.0.0.1 -l tar c sources/trimconfig-busybox \
  build/sources/{busybox,dropbear,strace}) | tar xv || exit 1

mv sources/* build/sources/* .
EOF
```

Getting data in/out of the emulator

- Note that $(netcat -l blah blah blah) must both launch the netcat server in the background and resolve to the port number on which the server is listening for an incoming connection for this to work. The "toybox" version of netcat does this.

Getting data in/out of the emulator

- You can have multiple "netcat dialing out to netcat" pairs and they'll trigger in sequence. The host version waits until the target makes a connection before running the appropriate command with stdin/stdout/stderr redirected to the network socket.

Getting data in/out of the emulator

- This does not require -redir, becaue the target is always dialing out to the host.
- Might need "killall netcat" to clean up if the target exits early and doesn't consume all the waiting netcat servers.

Getting data in/out of the emulator

- Build/install dropbear inside the emulator
  - A very nice self-contained embedded version of ssh/sshd.
  - Both programs in a 100k binary
  - Does the busybox swiss-army-knife trick of behaving differently based on what name you used to run it

Getting data in/out of the emulator

- Run sshd on host's loopback, then scp tarballs to/from 10.0.2.2.
- Or pipe tarball through ssh, ala:
  - **tar cz dirname | ssh USER@HOSTNAME tar xvzC destination**

Getting data in/out of the emulator

- Unlike Apache, sshd requires little configuration. Just install/run.
  - Can also be run without root access if you do it on a nonstandard port and explicitly tell it where to find its config files.

Getting data in/out of the emulator

- Install dropbear's sshd inside emulator and -redir a host loopback port to emulated system's port 22.
- You can tunnel rsync over this using the "-e" option:
  - **rsync --delete -avHzP -e "ssh -C -p 9876" source/. 127.0.0.1:dest/.**

Getting data in/out of the emulator

- Combining rsync with out-of-tree builds allows easy debugging
  - Incremental rebuilds. (Assuming the project's make dependencies work.)
  - If your project's build doesn't already support out of tree builds, try using "cp -sfR srcdir builddir" to populate a directory of symlinks.
  - To trim dead symlinks after rsyncing the package source and potentially deleting files, "find -follow -type l", and pipe any it finds to "xargs rm".

Getting data in/out of the emulator

- This technique also mitigates most of the problems with network filesystems
  - The symlinks are case sensitive, even if the filesystem they point to isn't
  - The local filesystem the build writes new files to has persistent timestamps with good granularity
  - Builds almost never modify files in their distro tarballs, and deleting any would remove the symlink
  - So you can export your working source via a network filesystem, but build in a temporary directory of symlinks on an ext3 image.

Getting data in/out of the emulator

- As always when working in multiple contexts, keep track of where your master copy is.
- Personally, we always treat the contents of the emulators as temporary, and have "master" directories on the host which we copy into the emulators.
- That way we can always garbage collect leftover images without worrying about losing too much.

Debugging

- gdbserver (part of the gdb package) allows you to debug things remotely.
  - Runs on the target to trace an executable
  - Lets you use gdb without the overhead of running gdb itself on the target, which isn't always even possible (it's a big piece of GNU bloatware).

Debugging

- gdbserver runs on the target to trace an executable, talks to gdb via a serial protocol.
  - Serial protocols tunnel well over ssh or netcat
  - Run QEMU to -redirect a port (such as 9876) to the host loopback
  - Inside QEMU: **gdbserver 10.0.2.15:9876 FILENAME**
  - Run gdb on host using the "pipe to process" option:
    - file FILENAME
    - target remote | netcat 127.0.0.1 9876
    - set height 0

Debugging

- Note that QEMU has a "-s" option that lets you attach gdb to the emulated **hardware**, the way a JTAG would
- Uses the same gdbserver protocol.
- This debugs the OS, not the applications.

Debugging

- Either way requires a host version of gdb that understands your target's instruction set.
  - Most distro versions configured host-only.

Interacting with upstream package maintainers

- You have a bug you would like fixed
  - Offering them a test environment in which they can reproduce the problem.
  - Offering a development environment in which they can test their fixes.
  - Self-contained QEMU development environment images small enough to download, running locally without root access are good for both.

Performance considerations

- Building under emulation is going to be slower than cross compiling. Accept that up front.
  - But how bad is it, and how can we make it suck less?

Performance considerations

- Some numbers we had lying around
  - Using a cheapish early 2007 laptop and comparing the native build of "make 3.81" versus building it under the now ancient QEMU 9.0.

Performance considerations

- Native Build
  - Extract tarball: Just over 1 second
  - ./configure: 15 seconds
  - Run make (-j 2): 4 seconds
  - Total: 20 seconds.

Performance considerations

- Native build under (now ancient) QEMU 9.0 armv4l emulation
  - Extract tarball: 5 seconds
  - ./configure: 2 minutes and 40 seconds
  - Run make: 2 minutes and 50 seconds
  - Total: 335 seconds

Performance considerations

- That's a big difference
  - 20%, 10%, and 2%
  - Respectively totaling 6% of native performance.

Performance considerations

- How do we speed it up?
  - Problems
    - Translated code runs somewhat slower than native code
      - Looks like around 20% of native speed in this case
      - Varies per target and per QEMU version

Performance considerations

- Launching lots of short-lived programs is slow
  - Translating the code takes time
  - Essentially a latency spike in faulting in pages
- Host is SMP, QEMU isn't

Performance considerations

- Options
  - Wait for Moore's Law to improve hardware
    - 6% of native performance would be 4 doublings, I.E. 6 years of Moore's Law
    - QEMU developers also improving the software
  - Make it scale to SMP or clustering so we can throw money at the problem
  - Be clever so it's faster even on a cheap laptop
  - Let's do this one.

Performance considerations

- The distcc acceleration trick
  - Use distcc to call out to the cross compiler through the virtual network.
  - Hybrid approach between native and emulated build
  - We want to leverage the speed of the cross compiler without re-introducing the endless whack-a-mole of cross compiling

Performance considerations

- distcc is a compile wrapper When told to compile .c files to .o files
  - calls the preprocessor on each .c file (gcc -E) to resolve the includes
  - Sends the preprocessed .c file through the network to a server
    - Server compiles preprocessed .c code into .o object code
    - Sends the .o file back when done
    - Copies .o file back through network to local filesystem
- When told to produce an executable
  - calls the local compiler to link the .o files together

Performance considerations

- How do we use it?
  - Run distccd on the host's loopback interface, with a $PATH pointing to cross compiler executables
  - Install distcc in the target, with "gcc" and "cc" symlinks to the distcc wrapper at the start of $PATH, configured with DISTCC_HOSTS=10.0.2.2
    - And g++/c++ symlinks for C++ support
  - Run build as normal

Performance considerations

- This moves the heavy lifting of compilation outside the emulator (where CPU is expensive) into the native context (where CPU is cheap)

Performance considerations

- Why doesn't this reintroduce all the problems of cross compiling.
  - No "two contexts" problem.
  - Header resolution and library linking done locally inside the emulator. Only one of each set of files.
  - make runs locally

Performance considerations

- ./configure runs locally
  - Still asking questions the local (target) system can answer.
  - Able to run the code it builds.
  - All the cross compiler does is turn preprocessed .c code into object files.
  - That's pretty much the one thing cross compiling can't screw up. (If it couldn't do that, we'd never have gotten this far.)

Performance considerations

- The build doesn't have to be cross-compile aware, or configured for cross compiling
  - No CROSS_COMPILER=prefix-, no $HOSTCC
  - As far as the build is concerned, it's building fully natively

Performance considerations

- FWL sets this up for you automatically. To do it by hand:
  - on host
    - **PATH=/path/to/cross-compiler-binaries /usr/bin/distccd --listen 127.0.0.1 --log-stderr --daemon -a 127.0.0.1 --no-detach --jobs $CPUS**

Performance considerations

- on target

```
mkdir ~/distcc
ln -s /usr/bin/distcc ~/distcc/gcc
ln -s /usr/bin/distcc ~/distcc/cc
cd ~/firmware
export PATH=~/distcc:$PATH
export DISTCC_HOSTS=10.0.2.2/$CPUS
./cross-compiler.sh armv4l
```

Performance considerations

- Using distcc to call out to the cross compiler turned the above 2:50 down to 24 seconds.
  - Factor of 7 speedup
- The above numbers are old, almost certainly no longer accurate, and may actually be tolerable as-is. But they can still be improved upon.

Performance considerations

- Further speed improvements
  - Low hanging fruit
    - Installing distcc was cheap and easy
      - They'd already written it, and it did exactly what we needed.
      - Any other low hanging fruit?

Performance considerations

- Throw hardware at the problem
  - More parallelism is the way of the future
    - Grab more computers to run distccd on (clustering).
      - This is what distcc was originally designed for anyway
    - Build a big SMP server, increase $CPUS
      - Even laptops are heading to 4-way, Moore's law isn't stopping.
    - We'll give examples later.

Performance considerations

- Problem: running make and preprocessing quickly become a bottleneck.
  - Varies per package, but an 8-way server might be a reasonable investment for building the Linux kernel.
    - 32-way, not so much.

Performance considerations

- QEMU's built-in SMP is useless
  - Fakes multiple CPUS using one host processor
  - Doing proper SMP in QEMU would involve making QEMU threaded on the host.
    - Also emulating inter-processor communication, locking, cache page ownership...
    - Generally considered a nightmare by the QEMU developers.
    - Don't hold your breath.

Performance considerations

- Tweak emulator
  - Faster network
    - The above benchmarks were done with a virtual 100baseT card (rtl8139) that maxed out at around 3 megabytes/second data transfer on the test system, and became a bottleneck.
    - The preprocessed .c files are very large, and must be sent both ways across the virtual network.

Performance considerations

- Modern default on x86 is gigabit ethernet (e1000) (as of QEMU 0.11.0). More efficient (faster even in emulator).
  - Try to configure your target to use this if possible.

Performance considerations

- Re-nice the QEMU process to -20 and distccd to 20
- If QEMU is the bottleneck, make sure it gets dibs on CPU time
- distccd nices its children down slightly by default

Performance considerations

- If you're actually **developing** packages, copy your new source snapshots in via rsync and built out-of-tree.
- Actually use make's dependencies for once to avoid rebuilding code that didn't change.
- Don't re-run ./configure when you don't need it.

Performance considerations

- Strategic use of static linking?
  - The OS can cache translated pages if it keeps the file MMAPed.
  - It tries to do this to optimize the host.
  - QEMU's translation overhead exacerbates normal disk fetch and cache fetch behavior patterns. Existing optimization techniques already designed to mitigate this.

Performance considerations

- Dynamically linked pages count as self-modifying code, forcing QEMU to retranslate the page. Also OS may discard and re-loads the page in response to memory pressure.
- Give emulator at least 256 megs of RAM.

Performance considerations

- Ok, so what's **not** low hanging fruit?
- Two main approaches:
  - Identify bottlenecks and make them less dominant.
  - Improving scalability, more parallelism

Performance considerations

- distcc only addressed the "make" portion of configure/make/install.
  - ./configure quickly comes to dominate
    - Make already spent most of its time in ./configure even on the host
    - This is becoming true in general
      - autoconf doesn't easily parallelize
      - Trying causes it to do extra work.

Performance considerations

- Exec-ing 8 gazillion dynamically linked executables has horrible cache behavior, even outside an emulator.
- So this is going to stay slow. What do we do about it?

Performance considerations

- Parallelize at the package level.
  - Fire up multiple instances of QEMU in parallel and have each one build a separate package.
    - This quickly becomes a package management issue.
      - Dealing with prerequisites involves installing and distributing prebuilt binary packages to the various QEMU nodes.
        - One "master node" drives builds, other QEMU nodes build a package at a time each, and then distcc daemons compile.

Performance considerations

- See "accidental distros" above.
  - Not easy to do this and stay orthogonal, but we can at least avoid reinventing the wheel.
  - We decided to leverage Gentoo's "portage" package management system for this.
    - It's got a big emphasis on building from source already.
    - Sketched out a clustering build extension to the Gentoo From Scratch project.
      - Nobody's stepped forward to fund it yet.
      - Advancing at hobbyist rate. Check back.

Performance considerations

- It's tempting to cache ./configure output (save config.cache)
- But the cure's worse than the disease.
- Just cache the generated binary packages

Performance considerations

- Autoconf was the bane of cross compiling, and it's the bane of native compiling too. Just less so.
  - Rant about configure and make becoming obsolete
    - Note that open source never really uses make's dependency generation.
      - “make all" is the norm, anything else is debugging.

Performance considerations

- Use a faster preprocessor (gcc -E or cpp stage)
  - The tinycc compiler has a very fast preprocessor which could probably be turned into a drop-in replacement for gcc's with a little effort.
    - Unfortunately, tinycc project more or less moribund.
    - Last serious progress in mainstream was before Fabrice Bellard left, around 2005.
  - Supports a very limited range of targets, need target-specific default preprocessor symbols. (run "gcc -dM -E - < /dev/null" to see the full list)

Performance considerations

- Add ccache package on top of distcc to cache preprocessor output?
  - Is it worth it?
  - Launching another layer of executable more expensive inside QEMU, need to benchmark.

Performance considerations

- Better wrapper
  - Installing distcc was easy, but distcc isn't perfect.
    - It only understands some gcc command lines, and when it can't parse it conservatively falls back to the native compiler, even when there's work to distribute.

Performance considerations

- ccwrap already has to parse gcc command lines more deeply than distcc does, to rewrite them for uClibc and relocation.
  - Possibly we could add distcc functionality to ccwrap?
    - Use existing daemons, or write our own?
  - Alternatively, upgrade distcc itself, but see next point

Performance considerations

- Fewer layers of wrapper
  - exec is expensive due to translation overhead
    - Remember the "slowing 20% to 2%" example, above.
    - Currently up to four layers: distcc->ccwrap->gcc->cc1
    - This is why adding ccache may not be a win.
      - Still need to bench it anyway.

Performance considerations

- ccwrap is small and lightweight, only a page or two needs to be faulted in. Trying to fix "gcc" to eliminate it not a big win.
- eliminating gcc's own wrapper instead, and teaching cpp/cc1/ld directly might be a bigger win.
- It does more work to accomplish less

Performance considerations

- Teaching ccwrap to do distcc might also be a win for this reason.
  - Teaching ccwrap to do ccache?
  - Complexity vs benefit.
    - Is this wheel improvable enough to be worth reinventing?

Performance considerations

- Sometimes gzipping data before/after sending it is faster than just sending it. (gzip is really cheap.) Need to benchmark this.
  - Helps "speeding up network" bottleneck.
  - pre-gzipped ccache data could be sent across the network without being re-compressed. THAT might be worth doing.

Performance considerations

- Persistent processes to avoid retranslation overhead?
- Leave a process running and pipe several files through it?
- Strategic use of static linking?

Performance considerations

- The OS can cache translated pages if it keeps the file MMAPed.
  - It tries to do this to optimize the host.
  - QEMU's translation overhead exacerbates normal disk fetch and cache fetch behavior patterns. Existing optimization techniques already designed to mitigate this.
  - Dynamically linked pages count as self-modifying code, forcing QEMU to retranslate the page. Also OS may discard and re-loads the page in response to memory pressure.

Performance considerations

- Feed data in larger chunks?
  - Lots of projects go "gcc one.c two.c three.c -c -o out.o"
  - distcc breaks 'em up into individual files, passes to separate daemons.

Performance considerations

- This has several downsides
  - Launches more individual processes
  - Gives the optimizer less to work with.
  - distcc is looking to maximize parallelism, but moving compilation outside of the emulator is our biggest win.
    - Our communications overhead is higher than normal.

Performance considerations

- Arranging for a fast host to run the emulator on.
  - Having a fast context in which to run builds doesn't prevent you from **also** running builds on your local laptop.
    - Something to ssh to when a local build would be cumbersome.

Performance considerations

- Building a fire-breathing server
  - Current sweet spot seems to be about 8-way server.
    - Both in terms of price/performance and in terms of build scalability.

Performance considerations

- We got a Dell server for ~3K in early 2009.
  - Tower Configuration (Can rackmount as 5U)
    - 8x 2.5Ghz (Xeon E5420)
    - 32GB RAM (mount -t tmpfs and build in that)

Performance considerations

- Amazon Cloud
  - Can rent an 8x server instance with 7GB of RAM for $.80/hour
  - Latency is an issue
  - Requires setup and teardown of instances
  - Potentially highly scalable though

Questions?

Rob Landley rob@impactlinux.com
Mark Miller mark@impactlinux.com

http://impactlinux.com