Introduction to cross-compiling for Linux

Or: Host, Target, Cross-Compilers, and All That

Host vs Target

A compiler is a program that turns source code into executable code. Like all programs, a compiler runs on a specific type of computer, and the new programs it outputs also run on a specific type of computer.[1]

The computer the compiler runs on is called the **host**, and the computer the new programs run on is called the **target**. When the host and target are the same type of machine, the compiler is a **native compiler**. When the host and target are different, the compiler is a **cross compiler**.[2]

Why cross-compile?

In theory, a PC user who wanted to build programs for some device could get the appropriate target hardware (or emulator), boot a Linux distro on that, and compile natively within that environment. While this is a valid approach (and possibly even a good idea when dealing with something like a Mac Mini), it has a few prominent downsides for things like a linksys router or iPod:

- **Speed -** Target platforms are usually much slower than hosts, by an order of magnitude or more. Most special-purpose embedded hardware is designed for low cost and low power consumption, not high performance. Modern emulators (like qemu) are actually faster than a lot of the real world hardware they emulate, by virtue of running on high-powered desktop hardware.[3]

- **Capability -** Compiling is very resource-intensive. The target platform usually doesn't have gigabytes of memory and hundreds of gigabytes of disk space the way a desktop does; it may not even have the resources to build "hello world", let alone large and complicated packages.

- **Availability -** Bringing Linux up on a hardware platform it's never run on before requires a cross-compiler. Even on long-established platforms like Arm or Mips, finding an up-to-date full-featured prebuilt native environment for a given target can be hard. If the platform in question isn't normally used as a development workstation, there may not be a recent prebuilt distro readily available for it, and if there is it's probably out of date. If you have to build your own distro for the target before you can build on the target, you're back to cross-compiling anyway.

- **Flexibility -** A fully capable Linux distribution consists of hundreds of packages, but a cross-compile environment can depend on the host's existing distro from most things. Cross compiling focuses on building the target packages to be deployed, not spending time getting build-only prerequisites working on the target system.

- **Convenience -** The user interface of headless boxes tends to be a bit crampled. Diagnosing build breaks is frustrating enough as it is. Installing from CD onto a machine that hasn't got a CD-ROM drive is a pain. Rebooting back and forth between your test environment and your development environment gets old fast, and it's nice to be able to recover from accidentally lobotomizing your test system.

Why is cross-compiling hard?

Portable native compiling is hard.

Most programs are developed on x86 hardware, where they are compiled natively. This means cross-compiling runs into two types of problems: problems with the programs themselves and problems with the build system.

The first type of problem affects all non-x86 targets, both for native and for cross-builds. Most programs make assumptions about the type of machine they run on, which must match the platform in question or the program won't work. Common assumptions include:

- **Word size -** Copying a pointer into an int may lose data on a 64 bit platform, and determining the size of a malloc by multiplying by 4 instead of sizeof(long) isn't good either. Subtle security flaws due to integer overflows are also possible, ala "if (x+y < size) memset(src+x,0,y);", which results in a 4 gigabyte memset on 32-bit hardware when x=1000 and y=0xFFFFFFF0...

- **Endianness -** Different systems store binary data iternally in different ways, which means that block-reading int or float data from disk or the network may need translation. Type "man byteorder" for details.

- **Alignment -** Some platforms (such as arm) can only read or write ints from addresses that are an even multiple of 4 bytes, otherwise they segfault. Even the ones that can handle arbitrary alignments are slower dealing with unaligned data (they have to fetch twice to get both halves), so the compiler will often pad structures to align variables. Treating structures as a lump of data that can be sent to disk or across the network thus requires extra work to ensure a consistent representation.

- **Default signedness -** Whether the "char" data type defaults to signed or unsigned varies from platform to platform (and in some cases from compiler to compiler), which can cause some really surprising bugs. The easy workaround for this is to provide a compiler argument like "-funsigned-char" to force the default to a known value.

- **NOMMU -** If your target platform doesn't have a memory management unit, several things need to change. You need vfork() instead of fork(), only certain types of mmap() work (shared or read only, but not copy on write), and the stack doesn't grow dynamically.

Most packages aim to be portable when compiled natively, and will at least accept patches to fix any of the above problems (with the possible exception of NOMMU issues) submitted to the appropriate development mailing list.

And then there's cross-compiling.

In addition to the problems of native compiling, cross-compiling has its own set of issues:

- **Configuration issues -** Packages with a separate configuration step (the "./configure" part of the standard configure/make/make install) often test for things like endianness or page size, to be portable when natively compiled. When cross-compiling, these values differ between the host system and the target system, so running tests on the host system gives the wrong answers. Configuration can also detect the presence of a package on the host and include support for it, when the target doesn't have that package or has an incompatible version.

- **HOSTCC vs TARGETCC -** Many build processes require compiling things to run on the host system, such as the above configuration tests, or programs that generate code (such as a C program that creates a .h file which is then #included during the main build). Simply replacing the host compiler with a target compiler breaks packages that need to build things that run during the build itself. Such packages need access to both a host and a target compiler, and need to be taught when to use each one. [4]

- **Toolchain Leaks -** An improperly configured cross-compile toolchain may leak bits of the host system into the compiled programs, resulting in failures that are usually easy to detect but which can be difficult to diagnose and correct. The toolchain may #include the wrong header files, or search the wrong library

paths at link time. Shared libraries often depend on other shared libraries which can also sneak in unexpected link-time references to the host system.

- **Libraries -** Dynamically linked programs must access the appropriate shared libraries at compile time. Shared libraries to the target system need to be added to the cross-compile toolchain so programs can link against them.

- **Testing -** On native builds, the development system provides a convenient testing environment. When cross-compiling, confirming that "hello world" built successfully can require configuring (at least) a bootloader, kernel, root file system, and shared libraries.

Footnote 1: The most prominent difference between types of computers is what processor is executing the programs, but other differences include library ABIs (such as glibc vs uClibc), machines with configurable endianness (arm vs armeb), or different modes of machines that can run both 32 bit and 64 bit code (such as x86 on x86-64).

Footnote 2: When building compilers, there's a third type called a "canadian cross", which is a cross compiler that doesn't run on your host system. A canadian cross builds a compiler that runs on one target platform and produces code for another target machine. Such a foreign compiler can be built by first creating a temporary cross compiler from the host to the first target, and then using that to build another cross-compiler for the second target. The first cross-compiler's target becomes the host the new compiler runs on, and the second target is the platform the new compiler generates output for. This technique is often used to cross-compile a new native compiler for a target platform.

Footnote 3: Modern desktop systems are sufficiently fast that emulating a target and natively compiling under the emulator is actually a viable strategy. It's significantly slower than cross compiling, requires finding or generating a native build environment for the target (often meaning you have to set up a cross-compiler anyway), and can be tripped up by differences between the emulator and the real hardware to deploy on. But it's an option.

Footnote 4: This is why cross-compile toolchains tend to prefix the names of their utilities, ala "armv5l-linux-gcc". If that was simply called "gcc" then the host and native compiler couldn't be in the $PATH at the same time.