Cross-compilation in pkgsrc

Taylor R. Campbell
riastradh@NetBSD.org

February 13, 2015

Abstract

When you want to compile applications for your feeble PowerPC network appliance, you don't want to run the compiler on the appliance: you want to use your 32-core Intel Xeon build machine.

Pkgsrc, the portable package build system, supports cross-compiling thousands of packages on NetBSD between any two CPU architectures. Provided a basic cross-compilation toolchain built by NetBSD's `build.sh tools`, pkgsrc cross-compiles the packages you ask it to build, natively compiling and installing any additional tools it needs.

Future work will enable cross-compiling between operating systems, not just between CPU architectures of a single operating system, and automatic building of the target OS's toolchain in the pkgsrc build process.

1 Introduction

Pkgsrc is a portable package build system: a repository of machine-executable descriptions of how to build various pieces of software from source and put them together into packages that can be installed and deinstalled, in the same way on a variety of Unix-like operating systems.

Normally, if you use pkgsrc on, for example, an x86 server running NetBSD 6.1, it will yield packages that can be installed on x86 machines running NetBSD 6.1. But building packages for the same CPU architecture and operating system as the compiler runs on makes for a painfully slow task of getting the packages you want for your tiny embedded PowerPC network appliance, which takes eons to run a C++ compiler.

NetBSD has supported cross-compilation as a first-class citizen for many years [2]: by default, when you want to build NetBSD, you first build a cross-compilation toolchain and then you compile NetBSD with it. The same process, of building a cross-toolchain and then compiling NetBSD with it, happens whether you are building NetBSD/amd64 on NetBSD/amd64 or building NetBSD/vax on an Apple PowerPC laptop running Debian. All builds of NetBSD published by the NetBSD release engineering team, for all CPU architectures and machine ports, are made on large x86 servers, so generating a new build of NetBSD/vax does not need to wait for an emulated or physical VAX machine.

2 Packages and dependencies

Pkgsrc consists of a large collection of package descriptions organized into categories, and some infrastructure to build packages out of them. A package description consists of a directory with a few files:

`DESCR` A human-readable description of the package.

`Makefile` A makefile, using pkgsrc's infrastructure, describing the package to a machine: what it is called, where its source distribution is located, how to build it, what other packages it needs, and so on.

`distinfo` Names, sizes, and hashes of all the files that pkgsrc must download in order to build the package.

`PLIST` A packling list, listing the files that the package installs.

There may be more files, such as a `Makefile.common` included by the makefiles in multiple package descriptions. From within a package description's directory, one can build and install the package with `make install`:

```
# $NetBSD: Makefile,v 1.7 2014/10/03 11:51:54 obache Exp $
#

DISTNAME=       libbind-6.0
PKGREVISION=    1
CATEGORIES=     net
MASTER_SITES=   ftp://ftp.isc.org/isc/libbind/6.0/

MAINTAINER=     pkgsrc-users@NetBSD.org
HOMEPAGE=       https://www.isc.org/software/libbind
COMMENT=        ISC Standard Resolver Library
LICENSE=        isc

CONFLICTS=              man-pages-[0-9]*

GNU_CONFIGURE=  yes
USE_LIBTOOL=    yes

.include "../../mk/bsd.prefs.mk"

.if !empty(USE_CROSS_COMPILE:M[yY][eE][sS]) && ${OPSYS} == "NetBSD"
# Yes, we'll have /dev/random on NetBSD, even autoconf you can't detect
# it by compiling and running a program when cross-compiling.
CONFIGURE_ENV+=         ac_cv_file__dev_random=yes
.endif

CONFIGURE_ARGS+=        --with-libtool=yes

MAKE_JOBS_SAFE= no

.include "options.mk"

.include "../../mk/bsd.pkg.mk"
```

Figure 1: An example pkgsrc makefile, with a workaround for a cross-compilation bug in the package.

```
# which scrypt
scrypt not found
# cd /usr/pkgsrc/security/scrypt
# make install
=> Bootstrap dependency digest>=...
=> Fetching scrypt-1.1.6.tgz
....
===> Install binary package...
# which scrypt
/usr/pkg/bin/scrypt
```

To avoid duplicating the work of building the same software over and over again on different computers, one can instead create a binary package from the package description with the `make package` command:

```
# which tmux
tmux not found
# cd /usr/pkgsrc/misc/tmux
# make package
=> Bootstrap dependency digest>=...
=> Fetching tmux-1.9a.tar.gz
...
# which tmux
tmux not found
# cd /usr/pkgsrc/packages/All
# ls tmux-1.9a.tgz
tmux-1.9a.tgz
# pkg_add ./tmux-1.9a.tgz
# which tmux
/usr/pkg/bin/tmux
```

The file `tmux-1.9a.tgz` is a binary package which can be installed without pkgsrc using the `pkg_add` command, on the same machine or another one. However, if on another machine, the `devel/libevent` package must be installed too, because the `tmux` program uses the library `libevent`.

Thus, `misc/tmux` is said to *depend* on `devel/libevent`, specifically as a *run dependency* because the latter must be installed at run-time if `tmux` is to function. Dependencies fall into one of three categories:

Run dependencies, or *full* dependencies, of a package are those that it needs installed at run-time in order to function. For example, software written in Python requires a Python interpreter to be installed in order run. A C program such as `tmux` that calls routines from a C library such as `libevent` requires the C library to be installed in order to run.

Build dependencies of a package are those that need to be installed at build-time as if for the target in order to build the package. A C program such as `xkbcomp` whose source code includes header files from a package such as `x11/xproto` seldom needs the header files to be installed in order for the program to function. If there is a library, such as `x11/libX11`, the C program will usually have a build and run dependency on the library, but it need not have a run dependency on a package containing only header files.

Tool dependencies of a package are those that need to be executed when building the package. For example, building `x11/libxcb` requires executing `xsltproc` to transform XML descriptions of the X11 protocol, so it has a tool dependency on `textproc/libxslt`. Programs that use `libxcb` need not transform XML, however, so there is no need for `textproc/libxslt` to be installed when these programs run.

(There are also *bootstrap* dependencies, for packages required in the operation of the pkgsrc infrastructure, but they are functionally little different from tool dependencies.)

When the user asks pkgsrc to build a package, pkgsrc will automatically build and install the tool, build, and run dependencies first, and when the user asks pkgsrc to install a package, pkgsrc will automatically install the run dependencies.

In addition to requiring that run dependencies be installed, if `tmux-1.9a.tar.gz` was built on an amd64 machine, it may be installed only on another amd64 machine — most binary packages contain compiled machine code for only one CPU architecture.

3 Cross-compiling packages

By default, binary packages built on a machine of one CPU architecture are fit to be installed only on other machines of the same CPU architecture, and the `pkg_add` tool will reject attempts to install them on others. However, pkgsrc can be configured so that instead of using the native C compiler, it uses a cross-compiler toolchain to build binary packages for a different CPU architecture. Pkgsrc relies on the NetBSD cross-toolchain, built with NetBSD's build.sh tools.[1]

Normally, the make variable MACHINE_ARCH in a pkgsrc makefile refers to the CPU architecture of the machine on which one is using pkgsrc, e.g. to run make install. When cross-compiling a package, MACHINE_ARCH is set to another CPU architecture, TOOLDIR is set to the NetBSD cross-compiler toolchain, and the resulting package will contain code for the specified CPU architecture.

If all package dependencies were run dependencies, that would be the whole story: if you ask pkgsrc to build a package on an x86 server with MACHINE_ARCH=powerpc, pkgsrc would simply build it and all its dependencies for PowerPC, and nothing would need to be installed until the system operator runs pkg_add. But build dependencies need to be installed, and tool dependencies include programs that need to be executed — building x11/libxcb requires processing XML documents with the xsltproc command from the textproc/libxslt package.

In that case, textproc/libxslt must be compiled natively, to run on the amd64 server, and installed before x11/libxcb can be built. And textproc/libxslt requires libgcrypt and libxml2 to be installed before it can be built or run, and requires them to be natively compiled too.

To accommodate these cases, when cross-building a package, pkgsrc will natively build its tool dependencies, and natively build any of the tool dependencies' dependencies, and so on recursively. Pkgsrc will also install cross-built packages not in their normal locations, but in a subdirectory called CROSS_DESTDIR, so that they do not interfere with the native packages: consider if the user asked to cross-build x11/libxcb and a package depending on textproc/libxml2, the first of which requires a native libxml2 and the second of which requires a cross-built libxml2.

A further complication arises when one of the tool dependencies is a compiler. Many compilers are capable of generating machine code for multiple different CPU architectures, but not at one time: when you build and install lang/gcc48, it will be configured for a particular CPU architecture, usually the one on which it was built. But to cross-compile one needs a compiler configured for a different CPU architecture.

To satisfy this, when building a tool dependency of a cross-built package for some CPU architecture, say PowerPC, pkgsrc will pass an additional setting TARGET_ARCH=powerpc to the tool dependency. Not all tool dependencies are configurable like this — for instance, xsltproc is the same whether you are going to use the XML for x86 or powerpc or arm — but packages have the option of paying heed to the target architecture.

In brief, before cross-building a package for, say, PowerPC, pkgsrc will:

- cross-build, but not install, all of its *run* dependencies;
- cross-build all of its *build* dependencies and install them relative to CROSS_DESTDIR; and
- natively build all of its *tool* dependencies, configured with TARGET_ARCH=powerpc and MACHINE_ARCH set to the native machine architecture, and install them relative to /.

4 Problematic packages

Many packages cross-build out of the box, especially ones that make simple uses of a C compiler. Some, however, do not. Common reasons for failing to cross-build include:

- A tool dependency recorded as a build dependency.

  In native pkgsrc builds, there is no difference between tool dependencies and build dependencies: in both cases, a package's dependency is built and installed before the package can be built. But for cross-builds, *tool* dependencies must be natively built and installed relative to /, while *build* dependencies must be cross-built and installed relative to CROSS_DESTDIR.

- The package's build system expects the same toolchain to generate programs that run natively during the build and to generate programs that can be packaged up for the target.

  But if you run a C compiler generating MIPS code on your x86 workstation, your x86 workstation will not natively run the resulting program.

  These cases can often be resolved by providing two C compilers to the package's build system: one that generates code that can run natively during the build, often called CC_FOR_BUILD, and

[1] Pkgsrc works on many operating systems, but the cross-compilation support is currently limited to NetBSD.

one that generates code that can be packaged up for the target, often called `CC`.

Some packages' build systems already support this, and it is a matter of teaching the pkgsrc makefile to set `CC_FOR_BUILD`:

```
CONFIGURE_ENV+= \
  CC_FOR_BUILD=${NATIVE_CC:Q}
```

Other packages need to be patched to distinguish native and target objects. The same principle applies to langauges other than C, of course.

- The package's build system expects to be able to run programs that query information about the target system, such as the existence of files, the order of bytes, and so on.

  Many programs configured with GNU autoconf do this. Since the target system is only a hypothetical while building a package, obviously this does not work in general, and if applied naively may yield the wrong answers: your x86 build machine's byte order is little-endian, but the PowerPC network appliance you're building packages for is big-endian!

  Fortunately, most of these queries can be answered in advance. For example, if you are building for NetBSD, the file `/dev/urandom` will exist, and if you are targeting PowerPC, the byte order will be big-endian. Many packages, especially ones written with GNU autoconf, can have these questions answered with environment variable settings such as `ac_cv_file__dev_urandom=yes`:

```
CONFIGURE_ENV+= \
  ac_cv_file__dev_urandom=yes
```

Some especially large and complicated build systems make disentangling these issues difficult, however. Perl and Python are two notorious examples of this, and are not yet cross-compilable in pkgsrc — although one can simply natively compile them on a machine of the target architecture, and then copy the resulting binary packages over to another machine to cross-build packages that depend on them.

5 Related work

A cross-toolchain is not the only way to cross-build packages: pkgsrc also supports using distcc, in which a machine of the target architecture runs the pkgsrc makefiles and talks to a usually much faster build machine to ask it to run a C compiler. This requires a machine of the target architecture and more administrative overhead, and is usually much slower since it still requires much work to be done on the target system and incurs the overhead of network communication.

Many commercially supported embedded platforms are developed with cross-compilers. Google's Android operating system for phones and tablets normally runs on ARM and MIPS systems, but the toolchain is a cross-toolchain that runs on x86. Android is not a general-purpose Unix system, though, and the software developed for it is not general-purpose Unix software.

OpenWrt [3] is a Linux distribution for a variety embedded platforms, particularly for many off-the-shelf consumer routers with MIPS and PowerPC CPUs and a small amount of RAM. OpenWrt has a repository of general-purpose Unix software that be cross-compiled and installed on the embedded platforms, including difficult but popular packages like Python and Perl. However, it is not intended as a general-purpose Linux distribution outside embedded platforms, so it does not share maintenance effort with, e.g., Debian. In contrast, pkgsrc works on a variety of operating systems and for a variety of purposes.

Various efforts have been put into cross-compiling FreeBSD ports [1], including running a native toolchain in the QEMU emulator. Recent work has been done to cross-compile FreeBSD ports to ARM, not with a cross-toolchain but by running a native toolchain in the QEMU emulator. Although running under QEMU may be faster on a large x86 machine than running natively on a tiny ARM board, it is much faster and more convenient still to run a cross-toolchain natively on the large x86 machine. None of these approaches appears to be supported well in the FreeBSD ports system.

6 Future work

Pkgsrc cross-compilation currently only works between different CPU architectures, not different operating systems or even different versions of the same operating system: you can set `MACHINE_ARCH` on NetBSD to compile for NetBSD on another CPU architecture, but you cannot set `OPSYS` and run pkgsrc on GNU/Linux to compile for NetBSD. Work is in progress to generalize pkgsrc cross-compilation from cross-CPU builds to cross-OS builds, but it is not yet in the main pkgsrc tree.

The user interface for asking pkgsrc to cross-build is klunky. For example, it requires the user to explicitly invoke NetBSD's `build.sh tools` to produce a cross-compilation toolchain. In principle, there is no reason it should not suffice to run

```
# make package \
  MACHINE_PLATFORM=NetBSD-7.1-powerpc
```

and have pkgsrc automatically build the toolchain, but this is not yet implemented.

Many, but not all, packages are cross-compilable. Many of the ones that are not cross-compilable are easy to fix with relatively simple patches. But there are a few major packages, such as Python and Perl, whose build systems make it actively difficult to cross-compile.

Pkgsrc's infrastructure for building many packages in bulk, pbulk, does not currently understand the difference between build and tool dependencies, so it is not yet easy to cross-build the entire pkgsrc tree at once on a fast, multi-core machine.

References

[1] Various ideas and attempts to cross build ports. FreeBSD Wiki page. `https://wiki.freebsd.org/CrossBuildingPorts` (retrieved 2014-12-05).

[2] Luke Mewburn and Matthew Green. build.sh: Cross-building NetBSD. *Proceedings of BSDCon '03*, 2003.

[3] OpenWrt. `https://openwrt.org/`.