Cross-building packages

Krister Walfridsson
`<kristerw@netbsd.org>`

29th September 2004

Abstract

It is often desirable to cross-build software, but most software packages have issues in their build process that needs to be corrected before it can be done. This paper describes a mechanism that automatically works around the issues, as implemented in the context of the NetBSD pkgsrc.

1 The basic idea

NetBSD runs on a wide range of architectures, and that cause several problems for the pkgsrc developers, since it is hard to test the packages on all architectures, or make the binary packages available for all of them.

People interested in binary packages often suggest introducing cross-compilation in pkgsrc. Their reasoning is usually that the GNU autoconf-generated `configure` scripts are designed to be used in cross-compiling environments, so at least the autoconf-using packages could be made cross-buildable. They are correct about the autoconf technology, but it is often not used in a cross-friendly way, so most packages would need modifications anyway. And doing even small changes to the 5000+ packages in pkgsrc is a daunting, and error prone, task.

One other approach is to build the packages in an emulator. This produces exactly the same binary packages that would have been built on the real hardware, without any need of modifying the packages. But this approach is slow (although building the package in an emulator running on modern hardware may be faster than building the package on the real machine for some of the older architectures...) There is, however, no need to emulate the complete hardware and operating system – only what is needed to build the packages. In particular, kernel mode does not need to be emulated, since the programs will not see any difference if the emulator does the equivalent action natively. This "as-is" rule can be applied on an even higher level: it is not necessary to run a program emulated if exactly the same result can be obtained by other means. For example, if the emulated environment tries to run

```
/bin/echo "Hello world!"
```

then it is not necessary to run `/bin/echo` emulated, since exactly the same result will be obtained (but much faster!) by running the command natively. And there is no need for the emulator to emulate running `gcc` when it can run a cross-compiler instead.

This makes it possible to cross-build packages relatively efficiently without doing any changes to the package. In fact, many packages build without emulating any program at all (more than possibly small test programs during configuration, but those are usually small enough that it only takes a couple of seconds).

There are however some packages that builds too slowly using this method, and those need to be modified (although it should be noted that they usually need less modifications than would have been the case for traditional cross-building methods). Experience with an actual implementation of these ideas shows that less than 1% of the packages need this kind of change in order to build efficiently.

The rest of this paper describes the implementation of these ideas.

2 How to build packages in pkgsrc

The NetBSD pkgsrc is a framework for building and managing third party software on a variety of operating systems (see `http://www.pkgsrc.org/` for a description of its features and inner working). For the rest of this paper it is enough to know that building a binary package (e.g. GNU Emacs) suitable for installing on other systems is as easy as

```
% cd pkgsrc/editors/emacs
% make package
```

This will fetch the source code (using ftp or http), apply pkgsrc specific patches, build the package, run tests to see that it works (if the original source code distribution includes such tests), and tar it up to a binary package.

Pkgsrc does also have a script to build all packages. This mechanism is most often used to find packages that have build problems, or to build the full set of binary packages for download from the NetBSD ftp. This "bulk build" is run by

```
% cd pkgsrc
% sh mk/bulk/build
```

The `build` script usually build only the packages that has been updated since the last run, and the packages that depends on them.

3 How to cross-build packages

The main goal for the cross-building framework has been to make it as easy to use as possible – both for the pkgsrc developers and the users that want to cross-build packages. Building packages are therefore done exactly as for native packages, although a wrapper is used (in exactly the same way that the standard NetBSD source code is cross-built) instead of the normal `make`.

```
% cd pkgsrc/editors/emacs
% nbsimmake-shark package
```

There is also a wrapper for `sh` so that bulk builds can be done:

```
% cd pkgsrc
% nbsimsh-shark mk/bulk/build
```

The file system for the target needs to be set up before starting to cross-build packages. This is done by packing up and configuring the standard NetBSD release in the same way that is done when setting up an NFS-mounted root. The cross-building framework does also need to be set up. These two steps can be done by installing the cross-building package for the target architecture.[1]

The rest of this paper will assume cross-building for NetBSD-1.6/shark, with the shark's file system located at `/emulroot/shark/`.

[1]These packages are not available from pkgsrc yet, but the software can be downloaded from `http://www.df.lth.se/~cato/crossbuild/`

4 How the cross-building framework is implemented

4.1 The starting point

The first step in the implementation was to find suitable emulators to build on. The GDB distribution has an extensive collection of emulators (which are called simulators in the GDB terminology) that emulates many of the architectures that NetBSD runs on, so this was seen as a good starting point.

The GDB powerpc simulator can even run NetBSD powerpc binaries by running the system calls natively in the same way as must be done for the cross-building framework! That NetBSD emulation is however rather limited, since it was developed for running the GCC test suite only, and the code has bit-rotted over the years, so the GDB NetBSD emulation code has not been used in this project.

There are some important architectures missing from the GDB distribution (such as m68k). It is however rather easy to add the NetBSD system call emulation to "any" emulator, since it is only the machine instruction responsible for entering system calls that is affected. The only thing that needs to be done is to write a small glue layer between the emulator and NetBSD code, to make it possible to read the parameters, return the result of the system call, and to read/write memory from the emulated machine.

4.2 Basic functionality

Simple system calls, such as `close`, are easy to run natively:

```
void do_close(void)
{
  int d = get_parameter(0);
  int status = close(d);
  write_status(status);
}
```

The first line gets the parameter that the emulated program provided to the system call. The next line calls the system call natively, and the last line modifies the emulated state so that the emulated program gets the result.

More complex system calls are done in essentially the same way. The only difference is that data may need to be copied between the emulated memory space and the host's memory space:

```
void do_write(void)
{
  int d = get_parameter(0);
  EMUL_ADDR buf = get_parameter(1);
  size_t nbytes = get_parameter(2);

  void* tmp_buf = xmalloc(nbytes);
  read_memory(tmp_buf, buf, nbytes);

  int status = write(d, tmp_buf, nbytes);
  write_status(status);

  free(tmp_buf);
}
```

Care need to be taken so that differences in endianness, or in the width of data types, are compensated for when moving structures between the different memory spaces, so arrays and structures need to be copied one field at a time:

```
void do_socketpair(void)
{
  int tmp_sv[2];
  int d = get_parameter(0);
  int type = get_parameter(1);
  int protocol = get_parameter(2);
  EMUL_ADDR sv = get_parameter(3);

  int status = socketpair(d, type, protocol, tmp_sv);
  write_status(status);
  if (status != -1)
    {
      WRITE_INT32(sv    , tmp_sv[0]);
      WRITE_INT32(sv + 4, tmp_sv[1]);
    }
}
```

Note that the above function takes advantage of fact that both the target and the host are running NetBSD – the `type` and `protocol` would need to be translated if the host were running some other operating system.

4.3 System calls working on files

Not all system calls can be done as straight forward as in the previous section. Consider for example running a program that opens a file through an absolute path:

```
open("/usr/include/machine/types.h", O_RDONLY);
```

It should open the file from the emulated machine's file system, and not the one from the host's file system. This means that the emulator must transform all absolute paths by appending `/emulroot/shark` before calling the native system call. Modifications are needed in the other direction too; system calls like `getcwd` does return a path of the form `/emulroot/shark/foo` that must have the prefix stripped before the control is returned to the emulated program.

4.4 `execve`

One other special case is the `execve` system call. A call like

```
execve("foo", argv, envp);
```

cannot be run natively as-is, but need to be transformed to

```
execve("emulator", new_argv, envp);
```

where `new_argv` is the `argv` where the arguments needed for the emulator to run the original program have been prepended.

But all programs do not need to be run emulated. Consider for example `/bin/echo`. This cannot do any "dangerous operations" like starting new programs or accessing paths not present on the command line that need to be transformed, so `execve` of `/bin/echo` may as well be run natively. This is true for many programs, although they may need some modifications of arguments. For example `/bin/cat` is safe too, but it may take file names as arguments, so

```
cat /path/foo
```

must be transformed to

```
cat /emulroot/shark/path/foo
```

before it is executed.

And some programs may safely be run, unless some special arguments are used. For example

```
/usr/bin/awk -F "'" '/^PACKAGE_VERSION=/ {print $2}' file
```

is safe, but

```
/usr/bin/awk -f foo file
```

is not. This means that a small parser must be implemented for each system binary so that the dangerous constructs can be transformed, or the command run emulated, as appropriate.

One important program in this category is `gcc`. We may greatly decrease the time needed to build a package by `execve` a cross-compiler instead of running `gcc` in the emulator.

One other thing to look out for is environment variables that may affect how the program works. For example, the `/usr/bin/install` may strip binaries when installing them, and the program to use for stripping is provided in the `STRIP` environment variable. The emulator must therefor check `STRIP` to see if it points to a program that can be run natively. The `/usr/bin/install` need to be run emulated otherwise.

Yet another issue that needs care are symbolic links. Note that all symbolic links made in the emulator must point to the full path within the real file system in order for the following to work

```
% ln -s /tmp/foo.c .
% gcc foo.c
```

This has the effect that some programs that otherwise would be able to run natively cannot do that. The most important example is `tar`. The good thing is that `tar` is rather efficient, so we do not lose much by running it emulated.

4.5 Removing one bottleneck

One of the goals of the cross-building framework is that it should be easy to maintain, so the original idea was to avoid building custom versions of components provided by the NetBSD distribution. Profiling did however show that a big amount of the time spent cross-building was spent in emulating `sh` and `make`, so it was decided to create special versions of those two programs to make them safe to run natively. The only difference from the original versions are that the cross-versions have modified all library calls with file parameters so that they are correctly transformed, and modified all calls to the exec-family of calls to use the same mechanism the emulator uses for `execve`.

This has the effect that there is a risk of the real and emulated `make` and `sh` binaries being out of sync, but the goal is to eventually get these changes into the real NetBSD distribution, so this is hopefully just a temporary problem.

4.6 Shortcuts

The goal of the cross-building framework is to cross-build packages – not to be able to run arbitrary programs from another architecture. This means that the emulation does not need to be perfect, as long as it does not affect the building of packages. It may even in some cases be easier to modify a specific package to build with a limited emulator instead of implementing all needed functionality in the emulator.

One trivial example of a limitation that is unlikely to affect the resulting packages can be seen by:

```
% nbsimsh-shark
$ mkdir /bin/foo
mkdir: /emulroot/shark/bin/foo: Permission denied
```

Observe that the error message contains the full path from the host operating system instead of the path that the emulated environment passed to `mkdir`. It is possible to prevent this kind of path leakage by improving the parameter checking done before running the native `mkdir`, to make sure that native execution always succeeds (or alternatively, parse and transform the output). But this kind of situation is not common when building packages, so it is much easier to change the affected packages, if such packages are found.

Many packages looks at the file and line information that `gcc -E` outputs, to e.g. build the dependency information for `make`. This cause problems when the cross-compiler is used, since absolute paths gets an `/emulroot/shark` prefix. This could of course be solved by always running `gcc -E` emulated, but that results in an unnecessary increase in build time for many packages. It is much better to solve this by treating paths such as `/emulroot/shark/foo` as `/foo` within the emulated environment, although this has the effect that it is not possible to have a directory called `/emulroot/shark` within the emulated file system...

5 Future work

The bulk build does currently need to be run with root privileges because many packages need to set user and group etc. on the files they create. It is however possible to use the emulator infrastructure to build packages as an unprivileged user.

The idea is to do all file operations as an unprivileged user, and to record the side effect of the "privileged" system calls (such as `chown`) in a log file. The log file is consulted every time a file system call, such as `stat` is called, so that the the correct information is returned to the caller. This will ensure that the resulting binary packages get the correct owner etc. for its files.

Some packages may however need to run e.g. SUID programs during its build process, and some of those may fail when not being run with the "real" user ID. Such packages need to be modified to build as an unprivileged user, but it is believed that only a few packages are affected.

Note that this could be used for unprivileged native builds too, by e.g. emulating an i386 target on an i386 host.