build.sh: Cross-building NetBSD

Luke Mewburn, Matthew Green

The NetBSD Foundation

lukem@NetBSD.org, mrg@eterna.com.au

Abstract

NetBSD has a cross build infrastructure which allows cross-building of an entire NetBSD release including bootable distribution media. The build process does not require root privileges or writable source directories. The build process works on many POSIX compatible operating systems. This paper explains the changes made to NetBSD to enable this build process, enumerates benefits of the work, and introduces future work enabling cross building of any software for NetBSD.

1. Introduction

NetBSD [1] is the most portable Unix operating system in common use. It is freely available and redistributable, and runs on a broad variety of platforms from modern desktop systems and high end servers that can build an entire release in less than an hour, to embedded systems and older machines that may take several days to build a release.

In late 2001, work began on changes to improve the ability of NetBSD to be cross built, especially an entire release. This system is referred to as "*build.sh*", because that is the name of the script that is the user-visible front-end to the infrastructure.

NetBSD 1.6 was the first release to be shipped with *build.sh*, and the Release Engineering group of the NetBSD Project took advantage of it to cross-build binary releases for 39 platforms on a near daily basis during the release cycle for NetBSD 1.6 [2]. Previous releases required access to each of the various platforms by release engineers, or co-ordination with developers with that hardware. While that method works for a moderate number of platforms (NetBSD 1.5 released binaries for 20 platforms), it does not scale, especially as the number of platforms in NetBSD is growing (54 as of June 2003).

2. Background

2.1. NetBSD

Since the NetBSD project was started in 1993 it has had a goal of being portable [3] to many target platforms. There has been significant effort in designing, implementing, and improving NetBSD to make it easier to "port" to a new target platform [4]. Device drivers are written in a way that permits easy sharing between platforms without unnecessary code replication [5].

The source code portability of NetBSD did not equate to "ease of use" when building the system on a host other than the target platform, or indeed, natively.

Prior to *build.sh* a NetBSD release for a given platform was built "natively" on that platform on a version of the operating system that was "close" to the target release. There were exceptions, but these alternate processes were not simple to use nor easily automated, and had a variety of other limitations which *build.sh* addresses.

build.sh offers a level of flexibility in building NetBSD that has not been addressed by other open source operating systems.

2.2. Cross compiling Unix

Unix was cross-compiled from the beginning, but when native hosting was available, that became the main development methodology and has remained so.

Cross-compilation is the technique of running programs on a "host" system to generate object code for a different "target" system. This has not been an easy task for most system builders and has generally not been integrated into operating system build processes as used by open source operating systems.

Freely available software projects such as GCC [6] have supported being cross-compiled for a long time, and GCC is part of the GNU toolchain which NetBSD uses and is heavily dependent upon for cross-compiling.

2.2.1. An introduction to cross-compiling

There are many parts to a full cross compiler environment. Besides the compiler itself, many others tools and files are required to create functional programs. Everything that a normal compiler needs must be present. For the GNU toolchain, this includes:

- The compiler - `gcc`.
- The assembler - `as`.
- The linker - `ld`.
- The "binutils"; `size`, `nm`, `strip`, `ar`, etc.
- Header files (provided by NetBSD).
- Libraries (provided by NetBSD and the GNU toolchain).

Following is a quick overview of how it all works. This is basically the same for any compiler; in this example the details are from the GCC C compiler:

1. The C compiler front-end `gcc` calls the C pre-processor `cpp` on an input source file, usually a ".c" file, producing a ".i" file. This is still valid C code but will now be devoid of C pre-processor directives. (Actually in modern GCC, the `cpp` pass is done inside the `cc1` pass to speed up the process and provide better error reporting. This process is largely invisible to the user.)

2. `gcc` calls the back-end `cc1` with the output of `cpp` producing a ".s" file. This is a assembler source file corresponding to the input C file.

3. `gcc` calls the assembler `as` with the output of `cc1` producing a ".o" file. This is an object file corresponding to the input assembler file.

4. gcc calls the linker `ld` with the output of `as` plus several other files (sometimes collectively called the "crtstuff"), to produce an executable.

   In addition to creating executables, archive and shared libraries are built. Archive libraries are usually created with the `ar` binutils program from object files. Shared libraries are created by calling `gcc` with the `-shared` option, which calls `ld` with various options to create a shared library.

`gcc`'s `-v` flag may be used to see exactly what external programs are called. For example, cross-compiling a simple NetBSD/sparc "hello world" C program on a NetBSD/macppc box gives:

```
what-time-is-love ~> /tools/bin/sparc--netbsde
lf-gcc -I/dest/usr/include -L/dest/usr/lib -B/
dest/usr/lib/ -v -save-temps -o hello.x hello.
c
Reading specs from /tools/lib/gcc-lib/sparc--n
etbsdelf/2.95.3/specs
gcc version 2.95.3 20010315 (release) (NetBSD
nb4)
 /tools/lib/gcc-lib/sparc--netbsdelf/2.95.3/cp
p0 -lang-c -v -I/dest/usr/include -isystem /de
st/usr/lib/include -D__GNUC__=2 -D__GNUC_MINOR
__=95 -D__sparc__ -D__NetBSD__ -D__ELF__ -D__s
parc__ -D__NetBSD__ -D__ELF__ -Asystem(unix) -
Asystem(NetBSD) -D__GCC_NEW_VARARGS__ -Acpu(sp
arc) -Amachine(sparc) -D__sparc hello.c hello.
i
GNU CPP version 2.95.3 20010315 (release) (Net
BSD nb4) (sparc-netbsdelf)
#include "..." search starts here:
#include <...> search starts here:
 /dest/usr/include
 /tools/lib/gcc-lib/sparc--netbsdelf/2.95.3/in
clude
 /tools/lib/gcc-lib/sparc--netbsdelf/2.95.3/..
/../../../sparc--netbsdelf/include
End of search list.
The following default directories have been om
itted from the search path:
 /tools/lib/gcc-lib/sparc--netbsdelf/2.95.3/..
/../../../include/g++-3
 /tools/lib/gcc-lib/sparc--netbsdelf/2.95.3/..
/../../../sparc--netbsdelf/sys-include
End of omitted list.
 /tools/lib/gcc-lib/sparc--netbsdelf/2.95.3/cc
1 hello.i -quiet -dumpbase hello.c -version -o
 hello.s
GNU C version 2.95.3 20010315 (release) (NetBS
D nb4) (sparc-netbsdelf) compiled by GNU C ver
sion 2.95.3 20010315 (release) (NetBSD nb4).
 /tools/sparc--netbsdelf/bin/as -32 -o hello.o
 hello.s
 /tools/lib/gcc-lib/sparc--netbsdelf/2.95.3/co
llect2 -m elf32_sparc -dy -dc -dp -e __start -
dynamic-linker /usr/libexec/ld.elf_so -o hello
.x /dest/usr/lib/crt0.o /dest/usr/lib/crti.o /
dest/usr/lib/crtbegin.o -L/dest/usr/lib -L/des
t/usr/lib -L/tools/lib/gcc-lib/sparc--netbsdel
f/2.95.3 -L/tools/sparc--netbsdelf/lib hello.o
 -lgcc -lc -lgcc /dest/usr/lib/crtend.o /dest/
usr/lib/crtn.o
```

The output shows the `cpp0`, `cc1`, `as` and `ld` programs being called in succession. An object file is created and the sundry "crtstuff" is added to the final `ld` line.

If `gcc` is passed the `-save-temps` option the output of each program will be saved rather than deleted after it has been processed. In the above example, the following files were created:

```
what-time-is-love ~> ls -l hello.?
-rw-rw-r-- 1 mrg mrg    57 Jun 5  2001 hello.c
-rw-rw-r-- 1 mrg mrg  7820 Jul 7 15:01 hello.i
-rw-r--r-- 1 mrg mrg   872 Jul 7 15:01 hello.o
-rw-rw-r-- 1 mrg mrg   408 Jul 7 15:01 hello.s
-rwxr-xr-x 1 mrg mrg 68638 Jul 7 15:01 hello.x
```

These are the source file and the `cpp`, `cc1`, `as` and `ld` outputs, respectively.

For a cross compiler the assembler, linker and binutils must also be "cross" tools. For the GNU toolchain these are normally called "$target-$tool". For example `powerpc-eabi-ld` is the linker for the "powerpc-eabi" target.

Each cross compile environment needs to provide header files and libraries appropriate to the target platform. These may be provided either as a static set of files or the build process may produce them as part of its bootstrap. The NetBSD process uses both of these techniques for header files and builds libraries from source.

2.2.2. Targets, hosts, and build hosts

One part of cross compilers that is often confusing is the difference between a "target", a "host" and a "build host". These are:

target	The platform the toolchain creates output for.
host	The platform that runs the toolchain.
build host	The platform that builds the toolchain.

The three normal ways to build the GNU compiler are:

1. A native build is when all three host types are the same:

   ```
   ./configure && make bootstrap
   ```

2. A normal cross compile is when the "build host" is the same as the "host":

   ```
   ./configure --target=powerpc-eabi && make
   ```

3. A "Canadian cross compile" is when "build host", "host", and "target" are all different:

   ```
   ./configure --build=i386-netbsdelf \
       --host=i686-pc-cygwin \
       --target=powerpc-eabi && make
   ```

   This strange beast is called a "Canadian cross compiler", because when a name was needed there were three political parties in Canada. It is most often found when dealing with pre-built toolchains. The above example would produce a toolchain that will run on a i686-pc-cygwin host (i.e., a Windows system with Cygwin installed) and will target embedded PowerPC platforms. The toolchain would be built on a NetBSD/i386 machine.

   A normal cross-compiler is required to build a Canadian cross-compiler.

Most people who build software use a native build in all cases. Usually people using cross compilers use the same "build host" and "host".

The host compiler is the compiler used to bootstrap the cross compiler, and to build any other host tools that need to run on the build host.

3. Feature set

The features of the *build.sh* system are detailed below. Other systems have addressed some of these issues but not all of them.

3.1. Cross compilation of a NetBSD release

This is a huge benefit to developers who wish to use NetBSD on a target platform (often an embedded system) but prefer to use a different host development system.

This feature is dependent upon toolchain support as described earlier.

Suitable build systems include:

- Current and previous NetBSD releases.
- Other Unix-like systems, such as Darwin (Mac OS X), FreeBSD, HP-UX, Linux, and Solaris.
- Windows with Cygwin.

3.2. Simplicity

build.sh needs to be simple to use while retaining enough flexibility to allow advanced users and developers to customize their build environment.

NetBSD has a finite amount of volunteer resources, so simplifying the build process reduces the support load.

3.3. Read-only source trees

The source tree can be read-only and can be shared between builds for different target NetBSD platforms. This permits building directly from read-only media or "mirrors" of source code without causing unnecessary mirroring conflicts. In this case the build output is written to a separate writable section of the file system.

There are no restrictions besides path-name length on the file system location of the source tree or the object tree.

3.4. Root privileges are not necessary

No root or other special privileges are required in order to build distribution media. Previous build processes required privileges to create tar files or file systems that contained devices, files owned by other users, set-ID files, etc.

Most other build systems require special privileges. For example, the Debian fakeroot [7] mechanism requires the host operating system to support shared object libraries with $LD_PRELOAD functionality (where a shared object is loaded after the main program but before the program's shared object dependencies), and this breaks the next requirement.

3.5. Avoid non-portable OS features

Prior to *build.sh*, the creation of NetBSD boot media required root privileges to create a "loopback vnode disk device" (`vnd`), install boot blocks, create a file system on the vnd, mount the file system, and create file system entries owned by the non-building user. Most other systems still build their distribution media in this manner.

build.sh does not require host operating system features such as: virtual disk devices, loopback file systems, chroot cages, shared objects/libraries, or dynamically loaded modules.

3.6. Segregated build tools

build.sh builds "host" tools to $TOOLDIR, which is a location separate to the host system's native tools. *build.sh* uses the tools in $TOOLDIR to build the NetBSD system installing to a separate $DESTDIR location. This removes the "chicken & egg" problem that can occur when the host platform's in-tree tools would need to be upgraded to build a newer source tree.

The older technique using the in-tree toolchain had the drawbacks:

- Without massive contortions, the build host had to be running NetBSD of the same architecture and similar software vintage.
- The in-tree toolchain required periodic updating.
- Updating the in-tree toolchain often required special handling that could not be easily automated. For example, at one time */usr/share/mk* needed `make`, and at another time `make` needed */usr/share/mk* to be updated.

3.7. Minimize impact on NetBSD source

There was minimal impact on the existing NetBSD sources and build infrastructure.

We did not want to change the NetBSD source tree to support being built by foreign compilers. Accordingly the host's native toolchain is only used to build the NetBSD host tools.

It was necessary to improve the portability of the NetBSD host tools to non-NetBSD platforms. These changes were not significant.

3.8. Don't special-case cross-compiles

It was a requirement that *build.sh* was usable for native builds as well as for cross compiling.

This feature is another component of the effort to reduce the support overhead of building NetBSD, whilst retaining our desired flexibility.

4. Results

4.1. Benefits

4.1.1. Regular automated builds

During the NetBSD 1.6 release engineering process, 39 platforms had near daily builds of the base operating system release, ensuring that at least all build problems were well known every day, and that up to date builds were available for testing by anyone. The majority (37) of these builds were performed on a dual processor AMD MP2000+ machine generously donated by AMD and Wasabi Systems, and the alpha and sparc64 platforms were built on Matt Thomas' CS20 alpha.

This contrasts with the NetBSD 1.5 release where 20 platforms had a binary release; all of them were built natively on the particularly port, requiring each port to have a dedicated person to build it, with 12 days time from when the first port was finished and available to the last port.

4.1.2. Simplicity of use

The new *build.sh* is very simple and easy to use, yet powerful. Anybody can build a kernel, userland or full release for most platforms on the fastest machine they have available.

4.1.3. More portable source code

It has identified every tool in NetBSD used for building. We know what tools we need and what features these tools must support. We had to make sure these tools were portable programs that would build and run correctly on other platforms.

4.1.4. Cross-platform builds

We no longer require running a particular version of NetBSD in order to be able to build it. Building NetBSD 1.6 from a NetBSD 1.5 machine is now supported, as is building from Solaris, Linux and more. Gone are the days of source build bootstrap problems that has plagued NetBSD since day one.

4.2. Costs

4.2.1. Teething problems

build.sh took a long time to settle down and be stable and useful. On the other hand, adding new platform support is no deal now, this is a one time cost that has been paid.

4.2.2. Cross-unfriendly software

Much software is written without consideration to cross compiling and thus makes it very difficult to cross compile. Not only the way the software itself is written, but the way the software is actually built. Many software build processes build helper programs to generate real code, and so these programs must not only work correctly for the target system rather than the native system, they must be built by the "host" compiler. The automounter `amd` was affected by this, as the build process assumed the local host architecture was the same as the program being built, causing a sparc binary built on i386 to incorrectly byte swap data into little endian for processing.

4.2.3. CPU endianness and word size issues

Endianness and CPU word size issues within the toolchain can cause problems. Due to bugs in the version of GCC that we use (2.95.3), NetBSD/alpha can not be cross compiled from a 32-bit host, and NetBSD/i386 can not be cross compiled from a 64-bit host.

Fortunately, we know that with GCC 3.3, NetBSD/alpha can be built on 32-bit platforms, and NetBSD's version of GCC will be upgraded to 3.3.x before NetBSD 2.0 is branched.

4.2.4. Overhead in converting a platform

Not every platform has been converted to *build.sh* yet. In some cases (e.g., pc532, playstation2) the in-tree toolchain does not support those platforms in the *build.sh* framework, whilst in others the platforms may not have had a complete build infrastructure before *build.sh* was integrated, and there were limited resources prior to the NetBSD 1.6 release.

4.2.5. Performance implications

There is a time cost in compiling the host tools, but it is not considered significant enough to pose a problem. For example, on an AMD XP2500+, the times for the various methods of building an i386 release are:

Time to build tools with *build.sh*: `./build.sh -U tools`	5m 6s
Time to build a release with *build.sh*: `./build.sh -U release`	1h 28m
Time unsuccessfully spent attempting to `make release` with the old build process on a two month old system, which failed due to problems with out-of-date tools.	> 2h

5. Implementation

The implementation of the *build.sh* infrastructure comprises of a variety of elements.

The build process uses various environment variables to control its operation, including:

TOOLDIR	Pathname to host tools for a given host platform.
DESTDIR	Pathname to built NetBSD system.
RELEASEDIR	Pathname to resulting NetBSD release files.

5.1. src/build.sh

build.sh is a Bourne shell script designed to build the entire NetBSD system on any host with a POSIX compliant Bourne shell in */bin/sh*. It creates a directory for various host tools uses to cross build the system (referred to as $TOOLDIR), creates a "wrapper" to `make` to pass in various settings, builds the host tools with the make wrapper, and uses the host tools to build the rest of the system, into a staging area referred to as $DESTDIR.

build.sh is very flexible and can be used to build an entire release, a specific kernel, just the tools or make wrapper, or even upgrade the installed system from a previously populated $DESTDIR.

The order of operation of *build.sh* is:

1. Validate arguments including the target platform (which defaults to the host platform) which is stored in $MACHINE, and the target machine (CPU) architecture which is stored in $MACHINE_ARCH.

2. Build a host binary of `make` and install as *$TOOLDIR/bin/nbmake*.

3. Create the "make wrapper" shell script that contains various (environment variable) settings which control the build and install as *$TOOLDIR/bin/nbmake-$MACHINE*. (This may be the last operation performed for this invocation of *build.sh*.)

4. Build the host tools to generate the binaries for the target $MACHINE and install under $TOOLDIR, with the executable host binaries available in *$TOOLDIR/bin*.

5. Perform any other operations requested using the $TOOLDIR host tools, including:

    - Build a NetBSD distribution into $DESTDIR.

    - Build all the kernels for a specific NetBSD release and package those and the contents of the $DESTDIR into a release under $RELEASEDIR.

    - Build a specific kernel.

    - Create the release "sets" from $DESTDIR or the source "sets" from the source tree.

    - Upgrade a directory (usually "/") from $DESTDIR.

Certain $MACHINE platforms support more than one $MACHINE_ARCH CPU architectures (a bi-endian processor or are processor with more than one word size are considered to be separate CPU machine architectures), and *build.sh* supports building the different target CPU architectures as separate releases.

5.2. src/BUILDING

In-tree documentation for *build.sh*.

5.3. src/Makefile

This Makefile at the top-level of the source tree contains various targets to facilitate building the entire NetBSD source tree, including:

build	Build the entire NetBSD system, in an order that ensures that prerequisites are built in the correct order.
distribution, buildworld	Perform `make build` and then install a full distribution into $DESTDIR, including the contents of *$DESTDIR/etc* and *$DESTDIR/var*.
release	Perform `make distribution`, then builds kernels, distribution media, install "sets", and then packages the system under $RELEASEDIR.
installworld	Install the distribution from $DESTDIR to $INSTALLWORLDDIR (which defaults to "/").

5.4. src/share/mk

To simplify the build process, NetBSD uses a library of `make` Makefile include files (with the suffix ".mk") in *src/share/mk*. This was inherited from the 4.3BSD Networking/2 and 4.4BSD Lite releases, and has been significantly enhanced in the decade since.

5.5. src/tools

Traditionally, BSD systems are built using in-tree tools, compilers, include files and libraries. This isn't usable for cross-compilation, and has many other problems which this new infrastructure fixes, as documented in section 3.

src/tools contains the `make` infrastructure required to build various host tools used during the build. These tools are built using an autoconf-built compatibility frame-work, and use "reach over" Makefiles into the rest of the NetBSD source tree to minimize replication of code. The source code for these host tools have been slightly modified to allow them to be built in as a normal NetBSD (target) program, as well as a host tool.

The host tools are installed into $TOOLDIR, and the BSD `make` ".include" infrastructure in *src/share/mk* selects the various host tools in preference to the "standard" versions. For example, for an i386 target $CC will be set to *$TOOLDIR/bin/i386--netbsdelf-gcc* instead of *cc*.

There is support for using an external toolchain rather than the in-tree versions of binutils and `gcc`. This is useful when there isn't yet support in the in-tree toolchain for a target platform.

The host tools currently consist of:

```
as asn1_compile binutils cap_mkdb cat cksum
compile_et config crunchgen ctags db dbsym
file gcc gencat groff hexdump install
installboot ld lex lint lorder m4 makefs
makewhatis mdsetimage menuc mkcsmapper
mkdep mkdep mkesdb mklocale mktemp msgc
mtree pax pwd_mkdb rpcgen sunlabel texinfo
tsort uudecode yacc zic
```

The majority of the host tools are installed with an "`nb`" prefix, to differentiate them from similarly named commands on the host. The exceptions are the GNU toolchain programs, which already have a name such as `i386--netbsdelf-gcc`.

5.6. METALOG support

Traditionally, `install` is run as root to install paths into the appropriate location under $DESTDIR (which defaulted to "*/*"), with the appropriate ownership and permissions.

This prevents non-root users from building a full distribution, as many files need specific permissions, such as setting set-user-ID root on */usr/bin/su*.

To solve this issue, we enhanced the specification file for the existing `mtree` tool to support a full path name (versus a context-sensitive relative path name), and referred to the result as a "metalog" entry. An example entry is:

```
./usr/bin/su type=file mode=04555 \
    uname=root gname=wheel \
    time=1057493599.102665
```

For a given build, a *METALOG* file is created, with a "metalog" line for each installed path name. The *METALOG* file is manipulated and parsed by various tools as necessary throughout the full build process.

`install` was modified to optionally install the paths as the current user and without any special permissions, and instead log the requested permissions to the *METALOG*. The ".mk" files in *src/share/mk* and a small number of special case Makefiles were all that needed to be modified to take advantage of this support in `install`.

`pax` was modified to support parsing a *METALOG* for the list of paths to add to an archive, and even add "fake" entries for devices which may not have been created in $DESTDIR. This is used to build *tar.gz* files which contain the correct ownership and permissions from a *METALOG* and $DESTDIR populated by a build by an unprivileged user. The scripts that create the "installation sets" were enhanced to parse the *METALOG* and invoke `pax` appropriately.

5.7. makefs

Various platforms use distribution media which require a ffs file system to boot from. Previously, NetBSD built these using a "loopback vnode disk driver" (`vnd`), which is not available on many other systems, and requires root privileges to mount and write to in any case.

`makefs` was added to create a file system image from a directory tree and optionally a *METALOG*. It is conceptually similar to `mkisofs`, a GPL-ed application which creates ISO-9660 file system images.

With `makefs`, an unprivileged user can create a ffs file system, complete with device nodes (with the latter being specified in a *METALOG*).

While `makefs` has been written in a manner that easily supports the addition of different file system types, it currently only supports creating ffs file systems, in either little or big endian (since NetBSD's ffs code supports opposite endian ffs file systems, c.f. the "FFS_EI" kernel option, and support in userland tools such as `newfs` and `fsck_ffs`). Support for other file systems such as iso9660, ext2fs, and FAT has been considered, but is not a high priority at this time.

The implementation of the ffs back-end re-uses a reasonable amount of the existing source code in the current NetBSD kernel implementation of ffs (in *src/sys/ufs/ffs*), but for simplicity, functions such as the block allocation code were re-implemented in a simpler manner. Various assumptions were made as part of this process, including that block reallocation would not be necessary, since the size of all files and directories (as files) is known at file system creation time.

While ffs was originally implemented as a user process before its integration into the 4.2BSD kernel, after two decades of kernel hacking it is heavily tied to the buffer cache and other kernel sub-systems, which makes it more difficult to use in a stand-alone program.

5.8. installboot

Most platforms need boot blocks on their distribution media, and these are installed with `installboot`. Previously, each platform had its own version of `installboot` in */usr/mdec/installboot*, which was compiled as part of, and heavily dependent upon, the kernel sources for that platform. They also required root privileges and generally required kernel support for specific disk-label `ioctl()`'s and only worked on actual disk devices.

/usr/sbin/installboot is a replacement for the machine-dependent versions of `installboot`. It can function on file system images, and doesn't need root privileges or kernel support for specific `ioctl()`'s to do so. The design of the boot blocks between the various platforms has been standardized as well, even if the actual implementation is different due to obvious differences in platform hardware.

All platforms that shipped with binary releases for NetBSD 1.6 (except i386) were converted to this new infrastructure. i386 was a special case due to the baroqueness of the implementation of `installboot` for that platform, but as the "cross build" host for the i386 release was a dual processor i386 box, it could run the tool "natively". This has been resolved in NetBSD-current.

On a related note, it is possible to make CD-ROMs that boot NetBSD on multiple platforms. For example, i386, sparc64, and macppc can boot off the same disk, and there's ample room for other platforms. NetBSD 1.6, with the 39 platforms that shipped with a binary release and associated source, fit on 4 CD-ROMs and booted on 9 of them.

5.9. src/etc/postinstall

`postinstall` is a script that checks for and/or fixes configuration changes that have occurred as NetBSD has evolved.

`postinstall` was added to the build system to detect, and in most cases, automatically fix, changes to configuration that must be performed due to software changes.

The tests that `postinstall` supports are:

postinstall	*/etc/postinstall* is up to date
defaults	*/etc/defaults* is up to date
mtree	*/etc/mtree* is up to date
gid	*/etc/group* contains required groups
uid	*/etc/passwd* contains required users
periodic	*/etc/{daily,weekly,monthly,security}* is up to date
rc	*/etc/rc** and */etc/rc.d* is up to date
ssh	`ssh` and `sshd` configuration update
wscons	`wscons` configuration file update
makedev	*/dev/MAKEDEV* is up to date
postfix	*/etc/postfix* is up to date
obsolete	obsolete file sets
sendmail	`sendmail` configuration is up to date

6. Future Work

6.1. xsrc

NetBSD does not currently support a cross-build of "xsrc" (our copy of X11R6 / XFree86 4.x), but this will be fixed eventually. XFree86 4.3 added its own support for cross-compiling. We may implement our own method of cross compiling X11 for better integration with our build system. XFree86's cross-build solution does not address all of our requirements, especially the removal of the need for root privileges.

6.2. pkgsrc

"pkgsrc" (the NetBSD packages collection) is not cross buildable. A smaller number of particular, probably smaller packages would probably fairly easy to get cross buildable, but the vast majority would each require significant effort.

There have been two suggestions to simplifying the solution to this problem:

1. Krister Walfridsson has proposed building the packages natively in an emulator, using optimizations such as only emulating user-mode, and capturing system calls and running those natively (after manipulating the arguments).

   An enhancement to that is to detect if an emulated program is `gcc` (for example), and invoking the host's cross-compiler natively for that case.

   The initial progress looks very promising, with an arm emulator compiling six times faster than the native platform.

   This proposal would be acceptable for solving the "xsrc" problem in section 6.1.

2. Use a tool such as distcc [8] to distribute the compilation of C or C++ code to (faster) remote systems, even if those systems are different to the current system, as long as an appropriate cross-compiler is available. This would still require the use of the native system for part of the build process.

7. Conclusion

build.sh has been an extremely useful solution to a variety of problems. It's now much simpler to build NetBSD releases from systems running earlier NetBSD releases, and even build on other systems such as Darwin / MacOS X, FreeBSD, Linux, and Solaris.

Generally, the support issues in NetBSD using *build.sh* have been less than for previous releases, especially considering the number of platforms now supported. Previously, it was extremely important to upgrade various in-tree tools, includes, and libraries in a specific order (that often changed) before completing the rest of the build. Now, a full build can be made without impacting the running system, and then an upgrade easily performed once a successful build is available.

The authors regularly use *build.sh* to cross-build entire releases on our (faster) alpha, i386, macppc, and sparc64 systems for platforms such as alpha, i386, macppc, pmax, shark, sparc, sparc64, and vax, as an unprivileged user using read-only source.

Acknowledgements

Todd Vierling wrote the original version of *build.sh*, and setup the *src/tools* framework.

Jason Thorpe does a **lot** of work in the toolchain and associated *build.sh* infrastructure.

Erik Berls wrote and maintains the autobuild script that is used on the NetBSD Release Engineering machines to automatically build multiple release branches for multiple target platforms on multiple build machines.

Wasabi Systems, Inc. funded a lot of the development of this work by paying the salaries of various developers.

FSF provides the GNU toolchain which is so nicely portable and cross compile friendly.

References

[1] *Welcome to the NetBSD Project*, `http://www.NetBSD.org/`

[2] *NetBSD Project Autobuild [for NetBSD 1.6]*, `http://releng.NetBSD.org/ab/B_netbsd-1-6/`

[3] *Portability and supported hardware platforms*, `http://www.NetBSD.org/Goals/portability.html`

[4] Frank van der Linden, *Porting NetBSD to the AMD x86-64: a case study in OS portability*, BSDCon 2002, `http://www.usenix.org/events/bsdcon02/full_papers/linden/linden_html/`

[5] Jason R. Thorpe, *A Machine-Independent DMA Framework for NetBSD*, Usenix 1998 Annual Technical Conference, `http://www.usenix.org/publications/library/proceedings/usenix98/freenix/thorpe.dma.ps`

[6] *Welcome to the GCC home page*, `http://gcc.gnu.org/`

[7] *Package: fakeroot 0.4.4-9.2* `http://packages.debian.org/stable/utils/fakeroot.html`

[8] *distcc: a fast, free distributed C/C++ compiler*, `http://distcc.samba.org/`