Cross-compilation and cross-configuration

Eero Tamminen
Veli Mankinen
Lauri Leukkunen
Erik Andersen

2003-07-08

Abstract

This document explains the problems in cross-compiling Open Source software for embedded Linux targets. It will explain why cross-compilation and cross-configuration are two distinct problems and that the latter one hasn't been yet solved satisfactorily. Document will then describe a bootstrap process and utilities which will help cross-compiling by solving the cross-configuration problem. It will also outline a project for making re-targeting of the cross-compilation extremely easy.

Contents

I Cross-compilation **4**

1 Audience **4**

2 Introduction **4**

2.1 Problems . . . 4

2.1.1 Configuring . . . 4

2.1.2 Installation . . . 5

2.2 Summary . . . 5

2.3 Current solution . . . 6

3 Proposed solution **6**

3.1 Sandboxing . . . 6

3.1.1 Linux From Scratch . . . 7

3.2 SDK sandboxes . . . 7

4 Implementation **10**

4.1 Test system . . . 10

4.2 Sandboxing problems . . . 10

4.3 Target environment . . . 10

4.3.1 Bootstrapping . . . 10

4.3.2 Other utilities . . . 11

4.3.3 Limiting configure and compilation . . . 13

4.3.4 Using X11 etc. services . . . 14

4.3.5 Conclusion . . . 14

4.4 CPU architecture transparency . . . 15

4.4.1 Miscellaneous binary formats . . . 15

4.4.2 Target device setup . . . 16

4.4.3 Fooling configure . . . 16

4.4.4 Conclusion . . . 16

4.5 No target device . . . 17

4.5.1 Target compilation without target device 17
4.5.2 CPU emulators 18
4.6 Making root image 18
4.7 SDK sandbox directory hierarchy 18
4.7.1 Further reading 19
4.8 Implementation notes 19

5 Future directions **20**
5.1 SDK configuration and management 20
5.2 SDK configuration options 21
5.2.1 Configuration notes 21

6 References **22**

II Appendixes **23**

A Dynamic vs. static linking **23**

B Gcc cross-dependency mess **23**
B.1 C-library kernel dependency 24

C chroot-uid.c **24**

D ARM identification for binfmt_misc **26**

E Document distribution **26**

F Change log **26**

List of Figures

1 SDK with sandboxes 6
2 SDK tools 7
3 Development process actions 9
4 Scratchbox component dependencies 12
5 Sandbox and development target interaction 15
6 Sandbox contents 20

Part I

Cross-compilation

1 Audience

This document is targeted at people cross-compiling lots of large Open Source software projects and who want to scale and generalize that effort. To see what we're aiming at, see the section 5.

2 Introduction

Usually people understand cross-compilation as compiling a program for a CPU architecture different from the one used on the machine where the compilation is done. This is accomplished using a cross-compiler toolchain and cross-compiled libraries and specifying these in the Makefiles that build the software.

This is easy with the low-level software and compilation tools because they are required to support cross-compilation. It's a very different matter with Open Source software in general, especially with end-user applications and e.g. GUI toolkits.

2.1 Problems

Most of the Open Source software projects use a 'configure' script to configure their software for compilation. This script is produced by tools called 'autoconf' and 'automake', which process 'm4' macros written by the application developer. The script will generate the Makefiles that are used for building the software and a 'config.h' header file containing defines for features found on the build system.

Configure is meant to ease configuring the software for compilation and its default assumption is that the software will be *run in the same environment in which it was compiled in* and *run from the place where it was installed to*.

2.1.1 Configuring

Autoconf provides application developers certain macros to check out features in the system[1]. The problematic ones are:

AC_TRY_RUN Tries to compile, link and run given test code for some feature. Test programs return zero for success.

[1] http://www.linuxselfhelp.com/gnu/autoconf/html_chapter/autoconf_6.html

AC_TRY_LINK Tries to compile and link test code for library function existence.

AC_CHECK_LIB Tries to compile and link test code using certain library.

AC_SEARCH_LIBS This does the same as AC_TRY_LINK, but 'configure' tries to do linking from all of the *system* library paths, not just from ones given in CPPFLAGS, CFLAGS and CXXFLAGS environment variables.

Most of the application developers haven't taken into account cross-compilation[2] when using these macros, so 'configure' ends up *cross-compiling test code and trying to run it on build host* which breaks the configuring. Configure can also find libraries, headers and versions of those that are only present on the build host, not on the target, which will fail either compilation of the program or running it on the target.

The worst thing is that you don't have any idea whether 'configure' found the correct values or not, unless you go manually through tens of thousands lines of output it produces (or the binary fails when it's run). Even if you've checked the standard things, application developers can and have written also their own m4 test macros and test programs for 'configure' to run.

2.1.2 Installation

When software is configured, the installation path is often written into sources. This installation path is often included also into numerous configuration files, dynamically loaded libraries etc. And the software doesn't work if it doesn't find it's configuration files in the specified place!

So... You usually need 'root' user rights to be able to install software into same path on the build host that it will be run on the target. Then you have in your build host system directories a program that doesn't run on that machine, and to test and run the program on the target device, you have to first track all of its files...

2.2 Summary

Configure itself is not a problem. It solves very well the problem it was designed for; configuring software for system of which the application developer (who made the software) doesn't know anything, but which the configure script can find out information about. The problem is how to limit and be sure about what it finds, how to let it test whether the found functionality actually works as expected and to make it install the software into correct place, both for the build host testing and the target device.

[2] On Unix systems higher level software is traditionally compiled natively.

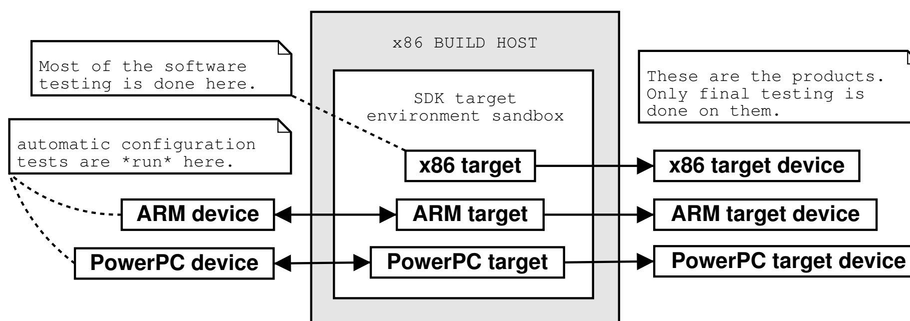

Figure 1: SDK with sandboxes

2.3 Current solution

Current solution in many projects using already existing Open Source software is to configure and compile it natively.

On real embedded development this solution is not optimal. It does not scale and embedded devices, even developer editions, are usually much slower than developer's x86 workstations and they have space constraints.

3 Proposed solution

3.1 Sandboxing

We have solved the problem by sandboxing the software configuration, compilation, building and testing into an environment that is exactly the same[3] as on target device.

Within the sandbox user can select the CPU target which software compiled inside the sandbox will see as the native one. This sandbox is our cross-compilation SDK[4]. It provides developer a self-contained environment containing everything needed for configuring, compiling and testing the software *on the target environment* on his *normal desktop*. Most of the time developers will need to use only the native (x86) CPU target which provides on the build host a Linux environment identical to that of the target device.

A company wanting to publish a SDK for it's device, can configure and compile this sandbox for their requirements and distribute it in binary format for the application developers. Sandbox can also contain device specific proprietary device drivers and libraries. In addition to the sandbox a complete company SDK will contain GUI tools, documentation etc.

[3]Where it matters to software that is being configured.
[4]SDK = Software Development Kit

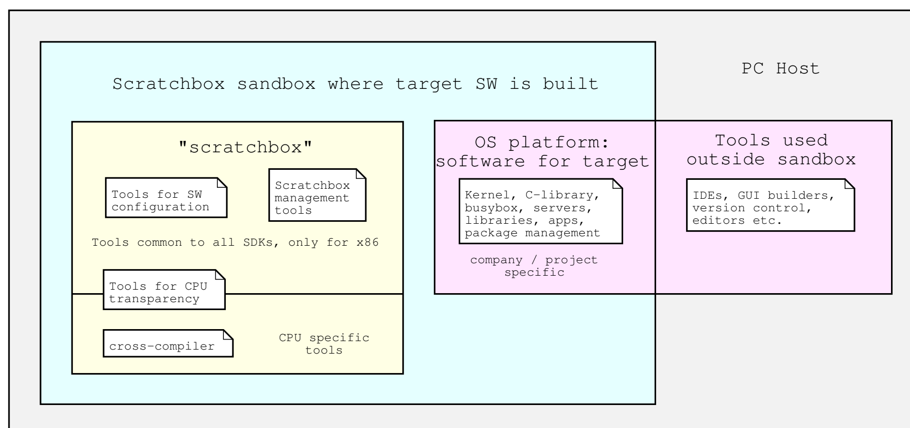

Figure 2: SDK tools

3.1.1 Linux From Scratch

"Linux From Scratch" project[5] has compiled a lot of information about making a (sandboxed) Linux distribution from scratch and provides that information also in a book form. "Target environment" section 4.3 will give an overview of these details. However, our main aim is *cross*-compilation and *cross*-configuration and we need to be able to use also other C-libraries than the standard glibc C-library so we have a few additional problems.

3.2 SDK sandboxes

Creation of SDK sandbox can be divided into following parts which are compiled and configured in successive steps:

1. Tools needed by configure scripts + CPU transparency + chroot + SDK configuration / package management. These are common to all SDKs and needed only for x86.
2. Cross-compiler. There has to be a separate compiler toolchain for each of the CPU target / kernel major version / C-library combination (see B).
3. Post-install part of the SDK tools.
4. Kernel / UML.
5. Debug tools[6] + *target* package management.

[5] http://www.linuxfromscratch.org/

[6] Some of the debug tools might be used outside the sandbox to connect to debug utilities inside the sandbox.

6. Building, testing and documenting target framework. This is often company specific.
7. Low level target software arbitrating hardware access. These can be e.g. Busybox, GUI servers (Nano-X, TinyX version of XFree86 etc).
8. Other target libraries and services included into SDK. Some of this “middleware” may be proprietary.
9. Normal software design, edit and version management tools used outside the sandbox.

First three steps bootstrap the SDK sandbox and produce what we call a *Scratchbox*[7]. Scratchbox utilities are compiled outside the SDK using the regular compiler (gcc) on the build host. Post-install part will be done when SDK is run the first time. Usually this is done by the person maintaining the SDK and normal developers get a fully installed binary version of the SDK.

After first three steps have been done, initial sandbox is ready for compiling rest of the software inside them, using the compiler(s) produced in step two.

Implementation section will give you more detailed explanation and reasoning for the above steps. Please get back here after reading it.

On build host All the targets inside the SDK sandbox use the same host-compiled utilities (needed by ’configure’ and other configuration utilities) which are run only on the build host. The compiler toolchain and libraries are specific to each of the sandbox target CPUs / C-library combination.

As the binaries produced by the PC target x86-toolchain can be run on your x86-desktop, we call this target the *native target.*

On target Target device will have two target systems:

- Root image and packages produced as the result of the cross-compilation and removing of unnecessary files. This is the system that will be put on target devices when they are ready. We call it the *product target system.*
- One used by the cross-compilation sandbox on the build host. This is used only in the product development, inside the sandbox. We call it the *development target system*.

Ideally you would have two target devices. One is used for developing the software and unit testing it with the *development target system* and second one is where you do all the integration and IPC[8] tests for the final system using the produced *product target system*.

Development target system is the one you use for *CPU transparency*.

[7] Linux From Scratch -sandbox
[8] Inter-Process Communication

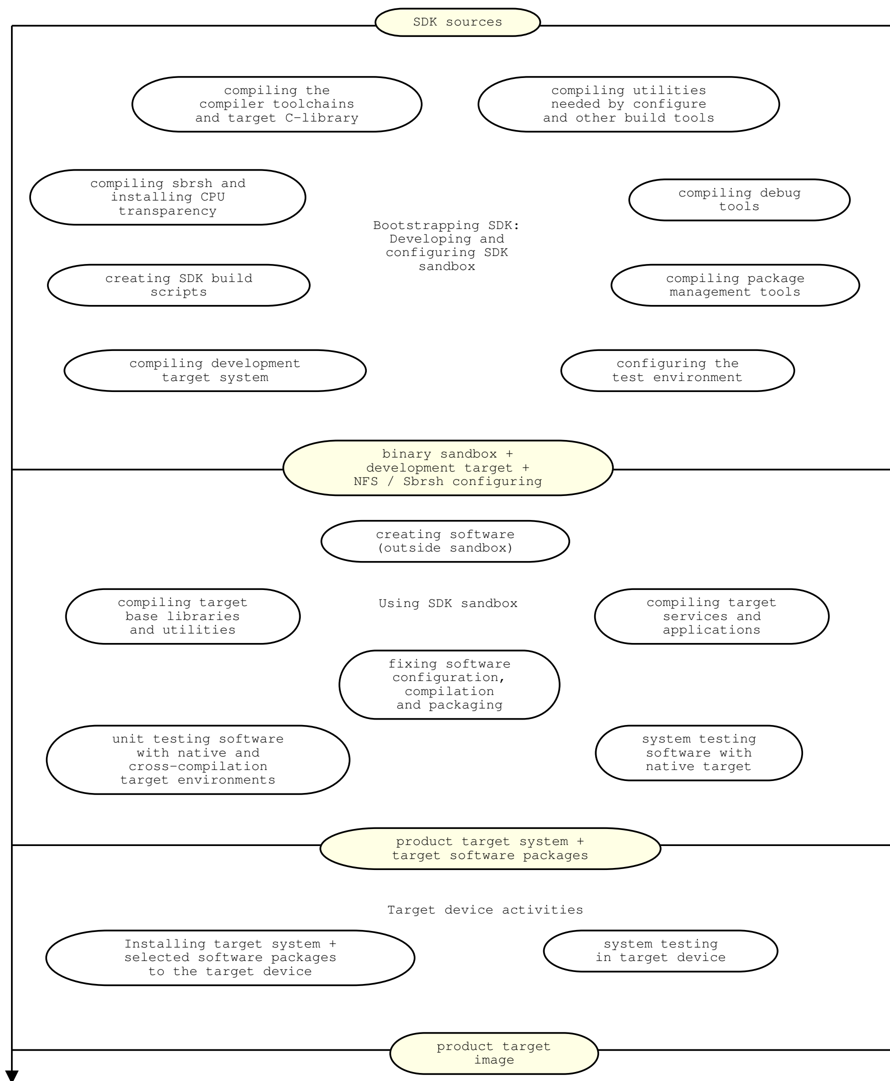

Figure 3: Development process actions

4 Implementation

4.1 Test system

Our build host machine is a x86 Linux desktop machine running RedHat 8 with glibc2 as the shared C-library and comes with all the normal desktop Linux libraries and utilities installed.

We're tried both uClibc and glibc as the shared C-libraries and dynamically linked Busybox for the command line tools. Target CPU is ARM.

4.2 Sandboxing problems

1. First problem of sandboxing 'configure' is that it needs a lot of (POSIX standard) software, which won't be available on the embedded target environment.
2. Second problem is that many 'configure' scripts test for features by compiling programs (with the supplied cross-compiler tool chain) and then running them. This doesn't work if program is compiled for a different CPU architecture than on which 'configure' tries to run it.

4.3 Target environment

4.3.1 Bootstrapping

First you have to compile *special* (see A) versions of the utilities that 'configure' needs for configuring the software on the *build host*. They need to be separately compiled because when you use them inside the SDK sandbox, they won't be able to use the normally installed libraries from your desktop. In Scratchbox this is accomplished by using a separate GCC spec file which tells the linker that the libraries for the utilities reside in a special directory which is same while you compile the tools outside the SDK and once you run them inside it.

For utilities which install a lot of external (e.g. configuration or locale) files, installation can be harder. They work inside the sandbox if they will be *there* in the *same* directory path as you had installed them on the *build host*. We used the `/scratchbox/` path as the installation prefix for these utilities on the build host and then copied this directory under the sandbox directory.

Normally 'configure' requires following standard GNU Unix utilities:

- gcc and C-library
- bash
- awk

- sed
- grep
- coreutils
- make

Busybox contains most of the things included into fileutils, textutils and findutils packages, but you still need to install at least following utilities from them: install, comm, split and find. If you don't use Busybox, then you need to install also following packages:

- diffutils

Autoconf and some of the configure files will additionally need:

- m4
- Perl

Perl is painful, because it has its own configuration system. Either it won't install or run correctly for uClibc, depending on whether you've compiled it outside or inside the sandbox. Its configuration system will need manual editing to get it working.

More complete dependencies between the tools are listed in figure 4.

4.3.2 Other utilities

Because also testing is done inside the sandbox target environment, developers will need to have debugging tools or stubs there. Some good tools are:

Strace A utility which traces for given program what kernel calls it does and what are their arguments and return values. This works almost on any CPU that Linux kernel work on.

GDB A symbolic source code debugger. You need to recompile your program to include debug information to fully benefit from GDB. For each CPU, you need to compile GDB server (containing network debug stubs) and normal x86-version of GDB which *understands that CPU*. Server is run inside the CPU specific sandbox or on the target, and GDB on the build host. This way you can have those *huge* debug binaries only on the build host and the GDB server on the target can use stripped versions of them. Additional benefit is that you can use any of the GUI GDB front-ends on the build host to run GDB.

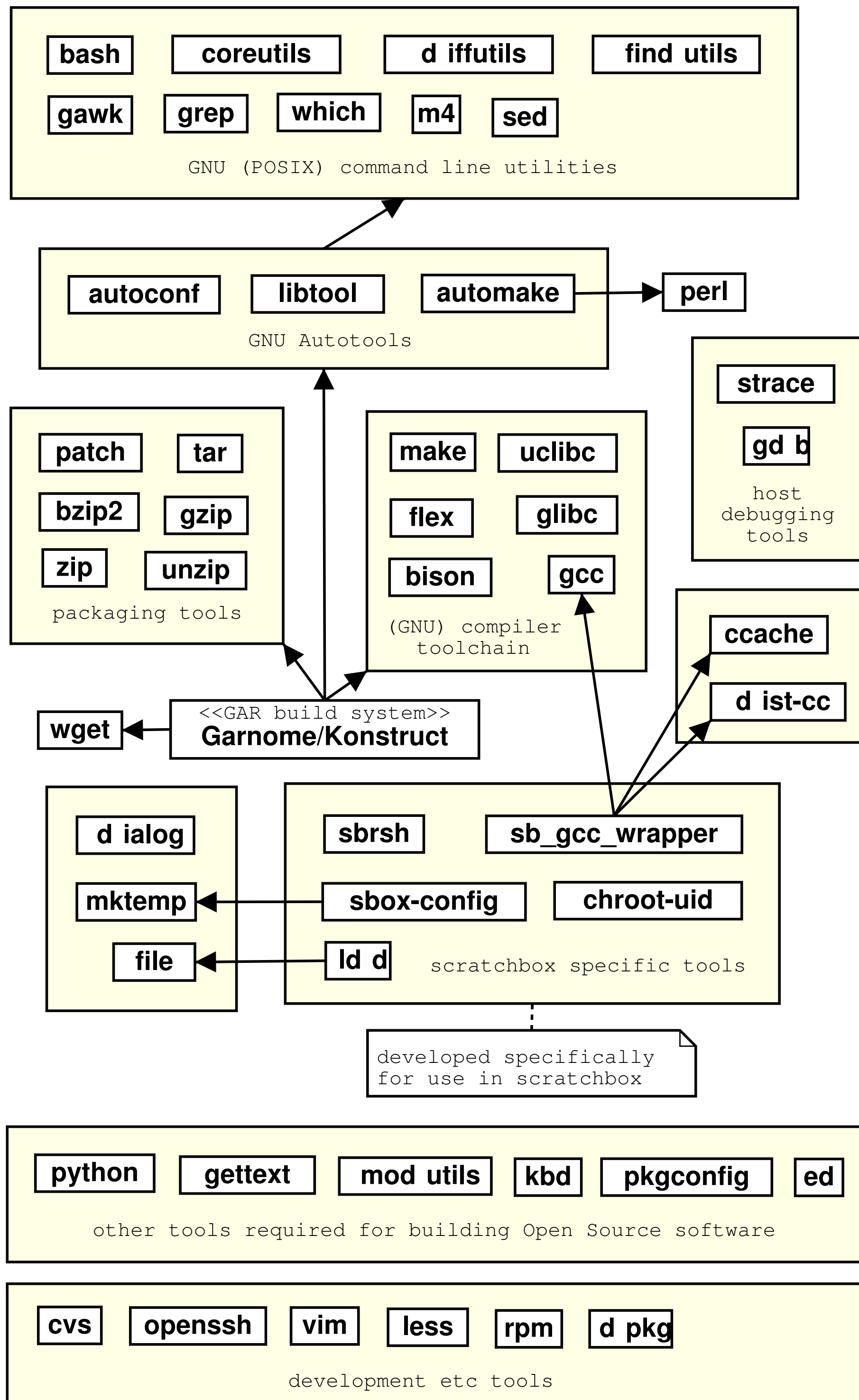

Figure 4: Scratchbox component dependencies

Valgrind A dynamic library for testing *all* kinds of programming errors[9] by running the program on a *simulated* x86-CPU. This is only for the x86 target as CPU simulation slows the program execution significantly and consumes more memory.

Because we *don't* want to configure, compile and maintain for each of the sandboxes CVS, IDE tools, GUI builders, code editors and other utilities used in creating and maintaining the source files, *these utilities are run outside sandbox* (see figure 2).

This means that sandbox needs an easy and synchronized way to access the sources you maintain outside the sandbox(es).

4.3.3 Limiting configure and compilation

There are two ways to limit the software configuration and compilation environment to the sandbox.

User Mode Linux UML, the User Mode Linux is a Linux kernel that can be run inside another Linux instance. You point it to an image file containing the root filesystem and it boots from there. It's a great way to test new kernel versions and debug drivers. In latest 2.5 Linux kernels UML is a standard compilation option.

In our case UML with the root image containing the above described "Target system" will provide the sandbox. Sandbox will also need to access the source files created and maintained outside it, this can be done by exporting the source directory with NFS from the build host, and mounting this inside the UML.

Currently UML has some minor problems and it's not included in the stable kernel. As UML is included in the 2.5 development kernel these issues should be fixed pretty soon. UML and NFS overhead will make compilation significantly slower.

Chroot Chroot is a kernel facility and user-space utility to limit program into given directory in the filesystem. Chrooted programs can't access *any*[10] files outside this directory, this includes devices and dynamic libraries.

If you don't need or want to use UML, you can use 'chroot' to limit the configure environment to what will be on the target device. Just 'chroot' the shell which you use to do the compilation. Using chroot is faster than using UML and SDK tools can be on a normal build host directory instead of a separate root image.

If you want to do *testing* also in the chroot environment, you need to be able to use device files. To accomplish this, you can either:

[9] Stack, heap and kernel call memory access errors, memory leaks, threading, bad x86-CPU cache utilization.

[10] Using symbolic links doesn't work because program can't access the file pointed by the link.

- 'mknod' the device files your target system requires into /dev/-directory in the sandbox. For the device major and minor numbers, look into the /dev/- directory of your build host, they are same on all the Linux versions. Note that several programs (e.g. ssh) will act funny if they can't access certain "system" devices like /dev/zero.
- Or just copy your whole /dev/ directory inside the sandbox. In Scratchbox /dev/ is copied inside the SDK and that copy is then mounted with --bind option inside to each SDK users' own Scratchbox sandbox.

If you want regular users to be able to use 'chroot' after you've setup the sandbox, install the chroot-uid.c program in the appendix C as suid root[11].

4.3.4 Using X11 etc. services

When doing testing in the sandboxed environment, you might need to access services such as X11 which are not inside the sandbox.

Easiest way would be to run Xnest with the same display attributes as your target system. Then you set the DISPLAY variable host part 127.0.0.1 and use same display number as you gave to Xnest. This should be as fast as using unix socket for the X11 communication.

User Mode Linux UML offers network interface to the host machine. You can use X11 through the network. The downside is that you can't use shared memory extension which will slow significantly image operations. See:
http://user-mode-linux.sourceforge.net/xtut.html

UML offers also access to devices in case you have full X11 inside your sandbox. For this the sandbox user will need access to the required devices...

Chroot With chroot you don't necessarily need to go through the network for service access, you just need to get the server Unix domain socket available inside the sandbox. Most of the servers put their socket into /tmp/ directory.

In Scratchbox we've mounted the system /tmp/ directory with the --bind option to each users own Scratchbox sandbox.

4.3.5 Conclusion

This will solve the first problem.

[11]Note that users can then use this program to fool other suid root programs. It's mainly intended for users who already have root privileges, but would rather do development as normal user... In the long run UML is better alternative, but not yet.

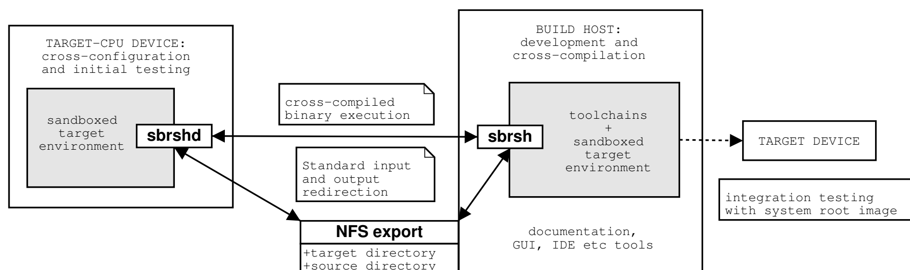

Figure 5: Sandbox and development target interaction

4.4 CPU architecture transparency

If you have a network enabled device with CPU compatible to your target device and enough memory, Linux provides a mechanism which can be used to make the CPU architecture transparent to the software we're configuring.

4.4.1 Miscellaneous binary formats

Linux kernel includes support for running miscellaneous binary formats transparently to the user and user-space programs. This support is in a module called *binfmt_misc* which can also be compiled statically to the kernel. You configure it by echoing following values to the '`/proc/sys/fs/binfmt_misc/register`' file:

- Name of the binary format
- How to identify the format
- "Interpreter" that should be invoked to run the binary

See D for an example.

In our case the "interpreter" will be a program that executes the binary on the target device and forwards the standard input, output and error streams, environment variables and execution error code between the build host and target. This program has to be compiled so that it works in the sandbox also when it's configured for the target CPU.

'ssh' can be used on build host and 'sshd'[12] on the target for doing above, but then you need to transfer the environment variables separately. To automate all this, we've developed '`sbrsh / sbrshd`' client-server software that automates all the required interaction (NFS mounting, environment variable transfer, I/O redirection, error returning, chrooting) between build host and target device.

You can get the binary CPU architecture signature for binfmt_misc configuration from the '`/usr/share/magic`' file used by the '`file`' utility.

[12]SSH keys required for automating this can be on the NFS directory and these keys shouldn't be used for anything else.

4.4.2 Target device setup

You need to have run-time environment similar to the sandbox on the target device (or device having the same CPU as the target). This includes the shell and dynamic libraries, not the utilities needed by the 'configure'.

This requires networking as you need to NFS-mount the directory where you're doing the source configuration on the build host, into same place (in directory hierarchy) on the target device as it's on the build host sandbox. This way target will have the same cross-compiled programs as build host, and any files that the executed program writes on the target device, will be in the same place on the build host sandbox.

This setup is called the "Development target environment".

4.4.3 Fooling configure

How the above works:

- 'configure' tests the system and won't find any libraries and include files located outside the sandbox.
- 'configure' creates and cross-compiles a test program and tries to run it.
- Kernel *binfmt_misc* module will note a registered binary format being 'executed', and runs it using the given interpreter.
- "interpreter" will run the binary on the target device from NFS mounted development directory and redirect all of it's input and output between target and build host.
- Result is that proper files are updated through NFS and program input and output forwarded correctly so that everything works for 'configure' as it were doing things natively.

4.4.4 Conclusion

This works both with the *chroot* and *User Mode Linux* and solves the second problem.

Because CPU is transparent to the sandbox, testing the cross-compiled binaries can be done in the pre-configured target CPU sandbox without manually logging to the target and copying the files. This requires NFS to be set up correctly (including the firewall settings of the host machine). When doing the tests[13], it's better check that you're doing them using the correct CPU target environment...

A single development target device can be enough for a whole developer group for the target configuration / compilation and testing phase.

[13] You can do only tests that don't require processes to communicate with each other this way. For integration tests you need the *target product system* environment.

4.5 No target device

You can do development for the *target environment* with SDK sandbox containing only the x86 target toolchain.

You need the target device and cross-compilation support *only* for testing *CPU architecture* specific features and of course making a target specific binary release of your software. For this it's enough that you have a device with the correct CPU, network interface supported by Linux and enough RAM. It doesn't need to be the *exact* device for which you're doing the development for, the same CPU (family) is enough (e.g. iPAQ for StrongArm target).

Some of the target specific kernel device driver testing can be done using x86 emulated device drivers in *User Mode Linux* and a x86-sandbox. This should be much easier than native testing when your final target has limited resources and/or no network.

4.5.1 Target compilation without target device

If you don't have a target device with networking capabilities or are otherwise lacking a target device, you can still do something with the software 'configure' scripts. Having the above described SDK will deal with the issues of 'configure' finding wrong libraries, versions, header files etc.

Then you have to fix just everything on configure scripts that tries to run cross-compiled binaries. Some possible ways of dealing with this:

- Ifdeffing (the m4 way) those lines in the scripts and providing the required values otherwise e.g. by using configure cache values and environment values.
- For certain things you can e.g. compile the programs on x86 and make configure to use your pre-compiled version of the test program. You still have to check the output for CPU architecture specific differences, with which our solution above deals with.
- Make the binfmt_misc solution (described in D) to catch all the cross-compiled program runs and return a correct value.

Here's an example of things you have to do for Glib2 to get it to cross-compile on a normal system:
http://bugzilla.gnome.org/long_list.cgi?buglist=77565
http://mail.gnome.org/archives/gtk-devel-list/2002-December/msg00057.html

In our opinion this is suitable way for fixing only a small number of software build systems.

4.5.2 CPU emulators

We tested two ARM emulators to see whether they could be used instead of a real device with target CPU:

GDB ARMulator GDB debugger contains an ARM emulator, see: http://www.uclinux.org/pub/uClinux/utilities/armulator/.

Swarm SoftWare ARM. Can run ARM binaries as you were running normal x86 binaries (i.e. doesn't need separate operating and disk images) but it's *very* slow, see: http://www.dcs.gla.ac.uk/~michael/phd/swarm.html.

Neither of these emulators could be used with the sandbox, either because their emulation of the target CPU was incomplete, they were too slow (several orders of magnitude) or they needed an environment which would have made the interaction with the sandbox environment too difficult.

For other CPUs (e.g. Motorola 68xxx family) there are other emulators, but we would assume the problems to be similar and in most cases our solution to be most convenient.

4.6 Making root image

TODO

4.7 SDK sandbox directory hierarchy

When you're cross-compiling for the target CPU, some of the binaries in the SDK sandbox are compiled for the *target device* CPU and some of the tools needed by the 'configure' are compiled for the *build host* CPU (in our case "build host" means x86 and "target" ARM binaries).

Here's is how we have divided the binaries in the *SDK sandbox* and what the different directories are supposed to contain:

/bin Contains SDK (x86) tools which are used by the configure and other build systems and used for configuring the Scratchbox.

/lib Dynamic and static system libraries for the *target* copied from the corresponding target compiler toolchain.

/usr *Everything* compiled inside the sandbox *for the target*, should be installed into */usr* directory. /usr is a link to target specific binary directory.

/host_usr Build-tools (built by target software packages) that are intended *not* to be run on the target.

/home The sources to configure and compile are under each developers home directory in the sandbox.

/etc Contains configuration files (user IDs, terminal and shell setting etc) for the *host tools*. Target software configuration files should go to /usr/etc/. When rootimage is is created, its /etc directory contents come from a special module that takes care of the target system setup.

/scratchbox SDK cross-compilers + build tools and the libraries & files required by them. These binaries can be run only on the host system. The dynamic library path in the binaries is hard-coded to /scratchbox/host_shared/lib/ directory (see A).

/targets Contains all the targets installed to the SDK. Top directories from the currently selected target directory are linked to the sandbox root directory.

/dev Mount –bind'ed from a copy of system /dev directory.

/proc Mounted inside SDK just like the system /proc.

/tmp Mount –bind'ed from the system /tmp so that sockets can be shared with services on the desktop.

Any other directories under the root directory contain target device specific files. It's possible to switch the target on the fly, this will just link directories to ones suitable for selected target. After changing target, the previous compiled object files etc. under */home* should be cleaned before starting to compile things for the new target.

The name of the current Scratchbox compilation target (CPU) is shown on the sandbox shell prompt.

4.7.1 Further reading

Everything on Linux should be using the "Filesystem Hierarchy Standard" for it's directory layout and file locations. You should also take a look at the "Linux Standard Base" documents. See the section 6 for links into these standards.

4.8 Implementation notes

- Currently we're concentrating more on using glibc than uClibc because uClibc internationalization and localization support is not yet (2002-11-30) fully compatible with common Open Source software supporting those features. Also, uClibc doesn't have a test suite with which it would be possible to compare it's API and performance (speed, memory use) objectively against the standard Linux glibc C-library. Other Open Source C-libraries are even worse in this respect.

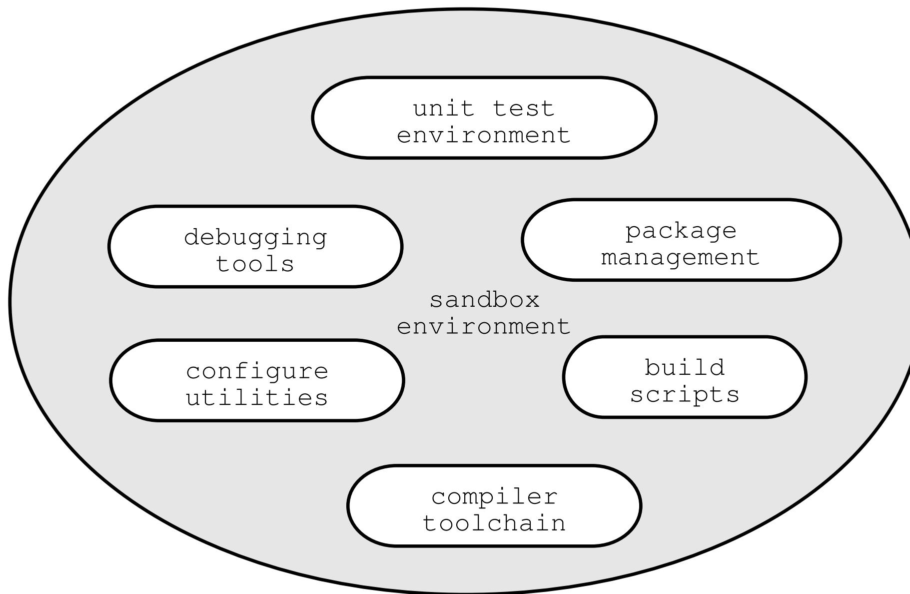

Figure 6: Sandbox contents

- You shouldn't spend serious effort in testing your software against non-standard C-libraries or other non-standard versions of standard system components *until* your software has been verified to work correctly. You do testing in an environment and with tools that are known to work together (Linux desktop, standard kernel, glibc, Valgrind etc). Debugging problems in your tools instead of problems in your own software is not productive. Don't do it until you have to.

5 Future directions

5.1 SDK configuration and management

Company or project specific SDKs can be based on our cross-compilation project (we've named it "Scratchbox") which contains the basic tools needed for configuring, compiling, testing and debugging standard Open Source software for different CPUs and C-libraries. These SDKs can then install their own extensions to the base using standard Open Source package management tools.

The base cross-compilation project can contain different sandbox configurations for different compiler and C-library combinations, chroot and UML kernels, windowing systems etc. It's important that others can contribute new base tools and configurations to the project. For this reason we've made compiling and configuring of the Scratchbox

SDK tools use the GAR ports system[14].

Currently package management isn't included *inside* Scratchbox, but it will in the future. Most likely it will be based on the Debian package format. We thought it also useful that developer can easily test his software with different CPUs, compiler versions and C-libraries. Therefore SDK has tools for developer to change the SDK base configuration inside the sandbox, before recompiling his software[15].

So, there will be two package management and configuration systems, one for installing the SDK for the host and configuring it, and another one used to configure and install software *inside* the SDK and on the target device.

There will be also binary releases of pre-configured and pre-compiled SDKs. These will be distributed as standard Linux distribution packages (tar.gz, RPM, dep etc).

5.2 SDK configuration options

Basic configuration options in the SDK will include selection of target CPU and C-library. The configuration utility will automatically switch the target environment directories for compiler, libraries, compiled binaries etc. according to the selected options.

Because we have control of all the utilities in this enviroment, it's easy to e.g. make all the compilations transparently use compiler cache or distributed compilation to speed it up. This could also be a SDK configuration option which you can enable when you have enough disk space.

Different projects can then add their own options for selecting higher level software for the target; windowing system, GUI toolkit etc.

5.2.1 Configuration notes

Kernel (driver) developers and application developers are very different people. They don't know about each others field of expertise (e.g. application developer doesn't know which memory settings should be used for certain target hardware *and* s/he shouldn't need to). For this reason an idea about having a unified configuration utility for the whole system is plain silly.

SDK should be modular enough so that each system level (kernel + C-library, servers, application middle-ware (framework) and applications) can be distributed separately as source for the developers of that level and as binaries for upper level developers. The source should of course be available to all (otherwise you would lose the code transparency advantage of Open Source), but it should be very easy get an already configured and compiled base system for your own work.

[14]GAR is used also to build development versions of Gnome and KDE desktops.
[15]Other alternative would have been to use separate SDKs for each target.

6 References

- GNU Autoconf: http://www.gnu.org/directory/autoconf.html
- GNU Automake: http://www.gnu.org/directory/automake.html
- GNU ftp site: ftp://ftp.gnu.org/
- User Mode Linux: http://user-mode-linux.sourceforge.net/
- User Mode Linux Community site: http://usermodelinux.org/
- User Mode Linux buildroot for uClibc: http://uclibc.org/cgi-bin/cvsweb/buildroot/
- Filesystem Hierarchy Standard: http://www.pathname.com/fhs/
- Linux Standard Base: http://www.linuxbase.org/
- Linux From Scratch: http://www.linuxfromscratch.org/
- Chroot jail project: http://www.gsyc.inf.uc3m.es/~assman/jail/index.html
- Systrace - System call Policy Generation: http://www.citi.umich.edu/u/provos/systrace/
- Familiar distribution: http://familiar.handhelds.org/
- uClibc C-library: http://www.uclibc.org/
- BusyBox site: http://www.busybox.net/
- GAR ports system: http://www.lnx-bbc.org/garchitecture.html

Part II

Appendixes

A Dynamic vs. static linking

First we though that by compiling SDK tools statically we would avoid the problem that dynamic C-library inside the SDK is different and that it can be for different CPU architecture than the one which SDK tools are compiled against.

However, that didn't work because glibc compiled programs can't be compiled completely statically, nss[16] and pam[17] functionality get their functionality from runtime loaded dynamic libraries which on their turn depend on dynamic glibc library. Recompiling glibc would have had the disadvantage that we don't know what name service and security module functionality developer would need in compiled into glibc and then SDK tool compilation would have included also glibc compilation which we would like to avoid.

Finally we solved this by forcing host linker to hardcode the dynamic library load path into SDK tool binaries and copying all the dynamic library dependencies (+ nss and pam modules) inside SDK into that hardcoded library location from the SDK build host. That way SDK tools don't find dynamic libraries intended for the target target software and target software configure scripts don't find host libraries.

B Gcc cross-dependency mess

Building of the base C and C++ libraries in integrated into GCC build. This is very awkward from the cross-compilation point of view because then *dynamic applications compiled with gcc will depend from the C-library against which gcc itself was compiled with*.

E.g. gcc v3.2 will compile stdlibc++ library (containing STL etc.) along with the g++ compiler. So, this library will now depend from the C-library with which g++ was compiled with and C++ applications compiled against that library will then also depend from C-library with which g++ was compiled with. Because libstdc++ is over 1MB, using it statically is not an option as libgcc.a is for gcc[18].

Because of above, you can't build compilers like this: compile gcc / g++ for each target CPU, compile C-libraries, compile C++ libraries, compile applications.

[16] Used e.g. resolving the user names and groups for 'ls'.

[17] Used e.g. when checking user password and other information in /etc/passwd.

[18] Why can't they release gcc base C-libraries as a separate package which is always released with the gcc package?

Instead you have to do it like this: compile first-phase gcc, compile C-library with that gcc version, re-compile whole gcc toolchain with the resulting C-library, then compile applications.

The result is that you need a separate toolchain for *each* CPU / C-library combination. It would have been nice to keep gcc toolchains as part of the SDK and only C / C++ -libraries target specific...

B.1 C-library kernel dependency

You should note that because C-library implements the kernel syscall interface, C-library always depends from the target kernel, or headers for the same (major) version of kernel. You can't just blindly use kernel headers from your host, Linux kernel user space API changes between major kernel versions.

So actually *you need a separate toolchain for each kernel major version / CPU / C-library combination...*

For 99% of the projects this is not a problem. For a generic cross-compilation SDK it is.

C chroot-uid.c

```
/* suid root'able version of 'chroot'.  See 'info chroot'.
 *
 * This binary has to be suid root as only root can use chroot() call.
 * 1. After that has been done, the real user ID and GID have to be
 *    restored so that user can't run anything with root priviledges.
 * 2. Then the user given program will be run chrooted.
 *
 * 2002 (w) by Eero Tamminen, Public Domain
 */
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(int argc, char *argv[], char *envp[])
{
  char *path, *prog, **args;

  /* check argument validity */
  if (argc < 3 || argv[1][0] != '/') {
    fprintf(stderr, "usage: %s   /absolute-path  path/program\n", *argv);
    fprintf(stderr, "  See 'man chroot'.  This is a version that can be run suid root\n");
    fprintf(stderr, "  as it restores real user ID before running the program.\n");
```

```
    fprintf(stderr, "  Path to the program should be relative to first path!\n");
    return -1;
  }
  path = argv[1];
  prog = argv[2];
  args = &(argv[2]);

  /* change to given directory and chroot to it */
  if (chdir(path) < 0) {
    perror("ERROR");
    fprintf(stderr, "chdir(%s)\n", path);
    return -1;
  }
  if (chroot(path) < 0) {
    perror("ERROR");
    fprintf(stderr, "chroot(%s)\n", path);
    return -1;
  }

  /* read _real_ user ID  and group and restore them */
  if (setuid(getuid()) < 0) {
    perror("ERROR");
    fprintf(stderr, "ERROR: setuid()\n");
    return -1;
  }
  if (setgid(getgid()) < 0) {
    perror("ERROR");
    fprintf(stderr, "ERROR: setgid()\n");
    return -1;
  }

  /* run the given program */
  if (execve(prog, args, envp) < 0) {
    perror("ERROR");
    fprintf(stderr, "ERROR: execve(%s, %s..., %s...)\n", prog, *args, *envp);
    return -1;
  }
  return 0;
}
```

D ARM identification for binfmt_misc

```
#!/bin/sh
# register 'arm_runner' to run ARM ELF binaries
echo ':arm_runner:M::\x7fELF\
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x28:\
\xff\xff\xff\xff\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
\xff:/bin/arm_runner:' > /proc/sys/fs/binfmt_misc/register
```

E Document distribution

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with no Invariant Sections, with no Front-Cover Texts, and with no Back-Cover Texts. A copy of the license is available at http://www.gnu.org/copyleft/fdl.html.

F Change log

2002-10-01 Started the document based on discussions with Veli and Lauri and emails with Erik.

2002-10-02 Completed most of the cross-compilation problems and rewrote the initial description of our solution based on Veli's feedback.

2002-10-03 Added bootstrapping section.

2002-10-09 Added references to Linux From Scratch project.

2002-10-10 Added UML and CPU transparency sections and a diagram about the sandbox and target interaction. Updated the document according to Veli's and Lauri's feedback.

2002-10-11 Added sections on debugging tools, sandbox directory hierarchy, future directions and information about scaling single development target device for multiple developers. Added *binfmt_misc* string for ARM binaries. Updated the target interaction diagram and added a diagram about the SDK.

2002-10-15 Added SDK sandboxes section according to Erik's suggestions and a section on using X11. Added chroot jail and uClibc UML buildroot references.

2002-10-16 Added development process & sandbox diagrams according to Erik's suggestions and a CPU emulators section.

2002-10-21 First stab at improving the structure of sandbox sections. Added a list describing steps in creating a SDK sandbox.

2002-10-31 Implementation name is changed to *scratchbox*. Added notes about GAR-system and a diagram of what is SDK, sandbox and scratchbox. Finalized the sandbox sections restructuring.

2002-11-01 Added SDK configuration notes to the "Future directions" section.

2002-12-04 Updated text and diagrams to how SDK now works: There are no multiple sandboxes, instead developer can install new targets inside the SDK sandbox and switch target inside the SDK. SDK tools are not compiled statically, they are compiled against dynamic host libraries which are copied into SDK directory where target software doesn't find them.

2002-12-17 Added a section on gcc, C-library and kernel cross-dependency mess.

2003-01-09 SSH/NFS based CPU transparency has been tested and found to be working but slow. It's problematic because there's no good way to transfer all the the environment variables so that it wouldn't break with multiple simultanous connections.

2003-01-30 Updated SDK diagrams.

2003-02-09 Generic talk about the SDK in FOSDEM 2003[19].

2003-03-21 SSH is replaced with home grown Sbrsh/Sbrshd.

2003-07-08 Updated the document and diagrams to reflect the latest Scratchbox state.

2003-08-04 Spell-checking and other minor corrections.

[19] http://www.fosdem.org/