Building and Using a Cross Development Tool Chain

Robert Schiele
`rschiele@uni-mannheim.de`

Abstract

When building ready-to-run applications from source, a compiler is not sufficient, but libraries, an assembler, a linker, and eventually some other tools are also needed. We call the whole set of these tools a development tool chain. Building a native tool chain to build applications for the compiler's platform is well documented and supported. As clusters become more and more widespread, it becomes interesting for developers to use the enormous CPU power of such a cluster to build their applications for various platforms by using cross development tool chains.

We describe how a development tool chain is structured and which steps have to be taken by its parts to build an executable from source. We also evaluate whether the characteristics of each step imply that a special version of this tool is needed for the cross development tool chain. Furthermore, we explain what has to be done to build a complete cross development tool chain. This is more involved than building a native tool chain, because intrinsic dependencies that exist between some parts of the tool chain must be explicitly resolved. Finally, we also show how such a cross compiler is used and how it can be integrated into a build environment on a heterogeneous Linux/Unix cluster.

1 Motivation

1.1 Unix Standard System Installations

Although in recent years some Unix vendors stopped shipping development tools with their operating systems, it is still quite common on most systems to have a C compiler, an assembler and a linker installed. Often system administrators use these tools to compile applications for their systems when binary packages are not available for their platform or when the setup of the binary package is not applicable to their local setup. For such scenarios, the system compiler is quite sufficient.

1.2 Development Usage

Although this so-called system compiler can also be used by a software developer to build the product he is developing on and is often done, this is in most cases not the best solution.

There are several reasons for not using the system compiler for development:

- In development you often have a large number of development machines that can be used in a compiler cluster to speed up compilation. Tools for this purpose are available, as `distcc` by Martin Pool, `ppmake` from Stephan Zimmermann with some improvements from my

side, or many other tools that do similar things. The problem is that when using the system compiler, you can only use other development machines that are of the same architecture and operating system because you cannot mix up object files generated for different platforms.

- As a developer, you normally want to support multiple platforms, but in most cases, you have a large number of fast machines for one platform, but only a few slow machines for another one. If you used only the system compiler in that case, you would end up in long compilation times for those platforms where you only have a few slow machines.

- Last but not least, you often also want to build for a different `glibc` release etc. than the one installed on your system for compatibility reasons. This is also not possible for all cases with a system compiler pre-configured for your system's binutils release and other system specific parameters.

1.3 Compiling for a Foreign Platform

We can solve all those problems by making clear to ourselves that a compiler does not necessarily have to build binaries for the platform it is running on. A compiler where this is the case, like the system compiler, is called a native compiler. Otherwise, the compiler is called a cross compiler.

We also need a cross compiler for bootstrapping a new platform that does not already ship a compiler to bootstrap a system with. But this cannot really be a motivation for this paper, as people that bootstrap systems most likely do not need the information contained in this paper to build a cross development tool chain.

In the following section we will show some basic principles of a development toolchain, how the single parts work and whether their characteristics require them to be handled specially when used in a cross development tool chain. In section 3, we will show what must be done to build a complete cross development tool chain and what are some tricks to work around some problems. In section 4, we show how to integrate the cross development tool chain into build systems to gain a more efficient development tool chain. Finally, we will find some conclusions on our thoughts in the last section.

2 How a Compiler Works

To understand how a compiler works and thus what we have to set up for a cross compiler, we need to have a look at the C development tool chain. This is normally not a monolithic tool that is fed by C sources and produces executables, but consists of a chain of tools, where each of these tools executes a specific transformation. An overview of this tool chain can be found in Figure 1. In the following, I will show those parts and explain what they do.

This section is not intended to provide a complete overview on compiler technology, but does only discuss some principles that help us to understand why cross development tool chains work the way they do. If you would like to have some detailed information about compiler technology, I recommend reading the so-called Dragon book [ASU86].

2.1 The C Preprocessor

The C preprocessor is quite a simple tool. It just removes all comments from the source code and processes all commands that have

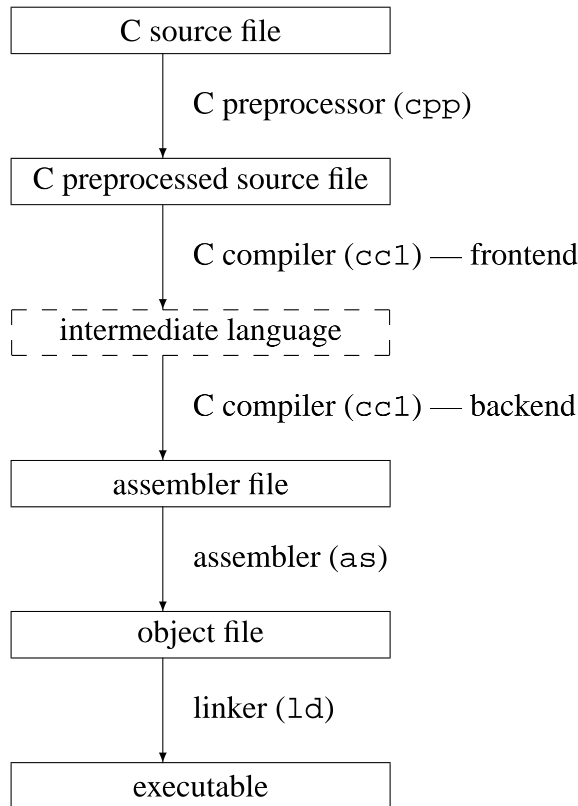

Figure 1: tool chain

a hash mark (#) on the first column of any lines. This means, for example, it includes header files at the position where we placed #include directives, it does conditional compiling on behalf of #if... directives and expands all macros used within the C source code. The output of the C preprocessor is again C source code, but without comments and without any preprocessor directive.

Note that most programming languages other than C do not have a preprocessor. It should be noted that preprocessor directives and especially macros make some hackers to produce really ugly code, but in general, it is a quite useful tool.

It can easily be seen that the C preprocessor itself should not be platform dependent, as it is a simple C-to-C-translator. But in fact, on most systems the preprocessor defines platform-specific macros like e.g. __i386__ on an ia32 architecture, and it must be configured to include the correct platform specific header files. Apart from that, in many compilers the preprocessor is integrated into the actual C compiler for performance reasons and to solve some data flow issues. Because of these reasons, the C preprocessor is actually not really platform-independent.

2.2 The C Compiler

The actual C compiler is responsible for transforming the preprocessed C source code to assembler code that can be further processed by the assembler tool. Some compilers have an integrated assembler, i.e. they bypass the assembler source code, but compile directly to binary object code.

We can divide the compiler into a front end and a back end, but you should note that in most cases these two parts are integrated into one tool.

2.2.1 The Compiler Front End

The front end is responsible for transforming the C source code to some proprietary intermediate language. This intermediate language should be ideally designed to be independent of both the source language and the destination platform to allow easy replacements of the front end and the back end. Because of that reason the front end is independent of the destination platform.

2.2.2 The Compiler Back End

The back end does the translation of the intermediate language representation to assembler code. As the assembler code is obviously platform-dependent, the back end is as well.

This results in the fact that although the front end is platform-independent, the whole C compiler is not because it is an integration of both the front end and the back end, where the latter is not independent.

2.3 The Assembler

The assembler is the tool that translates assembler code to relocatable binary object code. Relocatable means that there are no absolute addresses built into the object code, but instead, if an absolute address is necessary, there are markers that will be replaced with the actual address by the linker. The object code files include a table of exported symbols that can be used by other object files, and undefined symbols that require definition in a different object file. As both the input and the output of this tool is platform-specific, the assembler obviously depends on the platform it should generate code for.

2.4 The Linker

The linker can be considered the final part in the development tool chain. It puts all binary object code files together to one file, replacing the markers by absolute addresses and linking function calls or symbol access to other object files to the actual definition of the symbol. Some of those object files might be fetched from external libraries, for example the C library. We do not explain how linking to shared objects works, as it just makes things a bit more complicated, but does not make a real difference on the principles that are necessary to understand the development tool chain. The result of this tool is normally an executable. For the same reasons as with the assembler, the linker clearly depends on the destination platform.

More detailed information on the principles of linkers can be found in [Lev00].

3 Building the tool chain

As we now have some basic knowledge about how a development tool chain is structured, we can start building our cross development tool chain. We can find both the C preprocessor and compiler in the `gcc` package [GCC], which is the most commonly used compiler for Linux and for many other Unix and Unix-like platforms.

We use the assembler and linker from the GNU binutils package [Bin]. As an alternative linker for ELF platforms, there is the one from the elfutils by Ulrich Drepper, but this one is in a very early point in its life cycle, and I would not currently recommend using these tools for a productive environment. For the GNU assembler, there are also various alternatives available, but as changing an assembler does only a straightforward translation job and thus, no improvements of the results are to be expected, it is not worth integrating another assembler into the tool chain.

These are all tools for our tool chain, but we are still missing something: As every C application uses functions from the C library, we need a C library for the destination platform. We will use `glibc` [Gli] here. If we wanted to link our applications to additional libraries, we would need them also, but we will skip this part here. The essential support libraries for other `gcc` supported languages like C++ are shipped and thus built with `gcc` anyway.

The following examples are for building a cross development tool chain for a Linux system with `glibc` on a PowerPC. The cross compiler is built and will run itself on a Linux

system on an ia32 architecture processor. Although something might be different for other system combinations, the principles are the same.

3.1 The Binutils

The simplest thing to start with is the binutils package because they neither depend on the `gcc` compiler nor on the `glibc` of the destination platform. And we need them anyway when we want to build object files for the destination platform, which is obviously done for the `glibc`, but even `gcc` provides a library with some primitive functionality for some operations that are too complex for the destination platform processor to execute directly.

From a global point of view we have dependencies between the three packages as shown in figure 2.

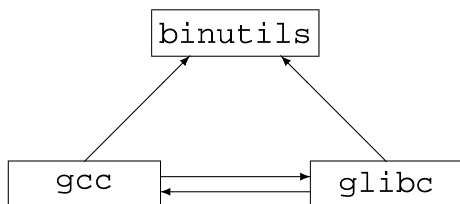

Figure 2: Dependencies between the packages

So we fetch a binutils package, unpack it and create a build directory somewhere—it is recommended not to build in the source directory—where we then call

```
../binutils-2.13.90.0.20/configure
    --prefix=/local/cross
    --enable-shared
    --host=i486-suse-linux
    --target=powerpc-linux
```

We set the prefix to the directory we want the cross development tool chain to be installed into, we enable shared object support, as we want that on current systems and we tell `configure` the host platform, i.e. the platform the tools are running on later, and the target platform, i.e. the platform for which code should be generated by the tools later. Afterwards, we run a quick `make`, `make install`, and the binutils are done.

As long as there is not a hard bug in the used binutils package, this step is quite unlikely to fail, as there are no dependencies to other tools of the tool chain we build. For the following parts we should expect some trouble because of intrinsic dependencies between `gcc` and `glibc`.

From this point on, we should add the `bin/` directory from our installation directory into `$PATH`, as the following steps will need the tools installed here.

3.2 A Simple C Compiler

Now we run into the ugly part of the story: We need a C library. To build it, we obviously need a C compiler. The problem is now that `gcc` ships with a library (`libgcc`) that in some configurations depends on parts of the C library.

For this reason, I recommend building the C library and all the other libraries on a native system and copying the binaries to the cross compiler tool chain or using pre-built binaries, if possible. If you build a cross compiler that compiles code for a commercial platform like Solaris, you have to do so anyway, as you normally do not have the option to compile the Solaris `libc` on your own. If you decide to build the C library with your cross compiler, continue here, otherwise skip to building the full-featured compiler.

We cannot build a full-featured compiler now, as the runtime libraries obviously depend on

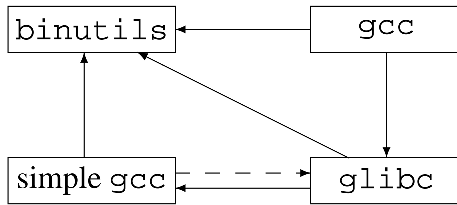

Figure 3: Dependencies with simple C compiler

the C library. This cycle in the dependency graph can be seen in figure 2. We can resolve this cycle by introducing a simple C compiler that does not ship these additional libraries, so that we get dependencies as shown in figure 3. But because of the reason mentioned above, for most configurations we cannot even build a simple C only compiler. That means we can build the compiler itself, but the support libraries might fail. So we just start by doing

```
CFLAGS="-O2 -Dinhibit_libc"
    ../gcc-3.2.3/configure
    --enable-languages=c
    --prefix=/local/cross
    --target=powerpc-linux
    --disable-nls
    --disable-multilib
    --disable-shared
    --enable-threads=single
```

and then starting the actual build with `make`. The `configure` command disables just everything that is not absolutely necessary for building the C library in order to limit the possible problems to a minimum amount. Sometimes it also helps to set the `inhibit_libc` macro to tell the compiler that there is no libc yet, so we add this also. In case the build completes without an error, we are lucky and can just continue with building the C library after doing a `make install` before.

Otherwise, we must install the incomplete compiler. In this case, the compiler will most likely not be sufficient to build all parts of the C library, but it should be sufficient to build the major parts of it, and with those we might be able to recompile a complete simple C compiler. We have to iterate between building this compiler and the C library, until at least the C library is complete.

The installation of an incomplete package can be either done by manually copying the built files to the destination directory, by removing the failing parts from the makefiles and continuing the build afterwards, or by just touching the files that fail to build. The last option forces `make` to silently build and install corrupted libraries, but if we have this in mind, this is not really problematic, as we can just rebuild the whole thing later and thus replace the broken parts with sane ones.

The simplest way of installing an incomplete compiler when using GNU `make` is calling `make` and `make install` with the additional parameter `-k` so that `make` automatically continues on errors. This will then just skip the failing parts, i.e. the support libraries.

3.3 The C Library

After having built a simple C compiler, we can build the C library. It has already been said that this might be necessary to be part of an iterative build process together with the compiler itself.

To build the `glibc` we also need some kernel headers, so we unpack the kernel sources somewhere and do some basic configuration by typing

```
make ARCH=ppc symlinks
    include/linux/version.h
```

Now we configure by

```
../glibc-2.3.2/configure
   --host=powerpc-linux
   --build=i486-suse-linux
   --prefix=
    /local/cross/powerpc-linux
   --with-headers=
       /local/linux/include
   --disable-profile
   --enable-add-ons
```

and do the usual `make` and `make install` stuff.

Note that the `-host` parameter is different here to the tools, as the `glibc` should actually run on the target platform and not, like the tools, on the build host. The `-prefix` is also different, as the `glibc` has to be placed into the target specific subdirectory within the installation directory, and not directly into the installation directory. Additionally, we have to tell `configure` where to find the kernel headers and that we do not need profiling support, but we want the add-ons like `linuxthreads` enabled.

In case that building the full `glibc` fails because building the C Compiler was incomplete before, the same hints for installing the incomplete library apply that where explained for the incomplete compiler. Additionally, it might help to touch the file `powerpc-linux/include/gnu/stubs.h` within the installation directory, in case it does not exist yet. This file does not contain important information for building the simple C compiler, but for some platforms it is just necessary to be there because other files used during the build include it.

After installation of the `glibc` (even the incomplete one), we also have to install the kernel headers manually by copying `include/linux` to `powerpc-linux/include/linux` within the installation directory and `include/asm-ppc` to `powerpc-linux/include/asm`. The latest kernels also want `include/asm-generic` to be copied to `powerpc-linux/include/asm-generic`. Other systems than Linux might have similar requirements.

3.4 A Full-featured Compiler

After we have a complete C library, we can build the full-featured compiler. That means we do now again a rebuild of the compiler, but with all languages and runtime libraries we want to have included.

With a complete C library, this would be no problem any more, so we should manage to do this by just typing

```
../gcc-3.2.3/configure
   --enable-languages=
       c,c++,f77,objc
   --prefix=/local/cross
   --disable-libgcj
   --with-gxx-include-dir=
       /local/cross/include/g++
   --with-system-zlib
   --enable-shared
   --enable-__cxa_atexit
   --target=powerpc-linux
```

and again doing the build and installation by `make` and `make install`.

4 Using the Tool Chain on a Cluster

We now have a full-featured cross development tool chain. We can use these tools by just putting the `bin/` path where we installed them to the system's search path and calling

them by the tool name with the platform name prefixed, e.g. for calling gcc as a cross compiler for platform powerpc-linux, we call powerpc-linux-gcc. The tools should behave in the same way the native tools on the host system do, except that they produce code for a different platform.

But our plan was to use the cross compiler on a cluster to speed up compilation of large applications. There are various methods for doing so. In the following we will show two of them.

4.1 Using a Parallel Virtual Machine (PVM)

We receive most scalability by dispatching all jobs that produce some workload to the nodes in the cluster. make is a wonderful tool to do so. A long time ago, Stephan Zimmermann implemented a tool called ppmake that behaved like a simple shell that distributed the commands to execute on the nodes of a cluster based on PVM. He stopped the development of the tool in 1997. As I wanted to have some improvements for the tool, I agreed with him to put the tool under GPL and started to implement some improvements. You can fetch the current development state from [ppm], but note that the documentation is really out of date and that I also stopped further development for several reasons.

If you want to use this tool, you just have to fetch the package, build it and tell make to use this shell instead of the standard /bin/sh shell by setting the make variable SHELL to the ppmake executable. Obviously you have to set up a PVM cluster before make this work. Information on how to set up a PVM cluster can be found at [PVMa]. To gain something from your cluster you should also do parallel builds by specifying the parameter -j on the make command line.

For example, if you had a cluster consisting of 42 nodes configured in your PVM software and ppmake installed in /usr/, you call

```
make -j 42
     SHELL=/usr/bin/ppmconnect
     ...
```

instead of just

```
make ...
```

CVS head revision replaced ppmconnect by the integrated binary ppmake.

There is also a script provided in the package that does most of these things automatically, but I do not like the way this script handles the process, so I do not use it personally, and such it is a bit out of date recently.

Note that there is a similar project [PVMb] by Jean Labrousse ongoing which aims at in integrating a similar functionality directly into GNU make. You may want to consider looking at this project also.

You should note that it is necessary for this approach that all files used in the build process are available on the whole cluster within a homogenous file system structure, for example by placing them on a NFS server and mounting on all nodes at the same place. Additionally, it is necessary that all commands used within the makefiles behave in the same way on all nodes of the cluster. Otherwise, you will get random results, which is most likely not what you want. This means you should always call the platform-specific compiler explicitly, e.g. by powerpc-linux-gcc instead of gcc, and the same releases of the compiler, the linker and the libraries should be installed on all nodes.

4.2 Using with `distcc`

The biggest disadvantage of the method described above is that it relies on central file storage and on identical library installations on all nodes. You can prevent these constraints at the cost of limiting the amount of workload that will be distributed among the nodes in the cluster to the compilation and assembling step. Preprocessing and linking is done directly on the system where the build process was started and thus not parallelized. Only compilation jobs are parallelized, all other commands are directly executed on the system, where the build process was invoked. Although this limits the amount of workload that really runs in parallel, this is in most cases not a real problem, as most build processes spend most of their time with compilation anyway.

The advantage of this approach is that you only need to have the cross compiler and assembler on each node. Include files and libraries are necessary only on the system on which the build is invoked.

Such an approach is implemented in Martin Pool's `distcc` package [dis]. This tool is a replacement for the `gcc` compiler driver. Preprocessing and linking is done almost in the same way the standard compiler driver does, but the actual compile and assemble jobs are distributed among various nodes on the network.

Although this solution obviously gives not the same amount of scalability, as not all jobs can be parallelized, it is for most situations a better solution, as from my experience it seems that many system administrators are not capable of installing a homogenous build environment on a cluster of systems.

5 Conclusion

Finally, we can conclude that it is not really difficult to build and use a cross development tool chain, but in most cases, building the whole tool chain is not as simple as described in the compiler's documentation because building cross development tool chains is not as well tested as building native tool chains are. Thus, you should expect numerous minor bugs in the code and in the build environment. But with some basic knowledge about how such a system works and, thus, what the source of those problems is, in most cases they can be easily fixed or worked around.

At least if you have an amount of systems for office jobs idling almost all of their time, it is worth investing some time for building up such an infrastructure to use their CPU power for your build processes.

As this is a tutorial paper, its contents are intended for people that do not have extensive konwledge on the topic described to help them understanding it. If you think something is unclear, some information should be added or you find an error, please send a mail to `rschiele@uni-mannheim.de`.

References

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[Bin] GNU Binutils. `http://sources.redhat.com/binutils/`.

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

[GCC] GCC Home Page—GNU Project—Free Software Foundation (FSF). `http://gcc.gnu.org/`.

[Gli] GNU libc. `http://sources.redhat.com/glibc/`.

[Lev00] John R. Levine. *Linkers and Loaders*. Morgan Kaufmann Publishers, 340 Pine Street, Sixth Floor, San Francisco, CA 94104-3205, 2000.

[ppm] SourceForge.net: Project Info—PVM Parallel Make (ppmake). `http://sourceforge.net/projects/ppmake/`.

[PVMa] PVM: Parallel Virtual Machine. `http://www.epm.oml.gov/pvm/`.

[PVMb] PVMGmake. `http://pvmgmake.sourceforge.net/`.