Toolchain options in 2023:

What's new in compilers and libcs?

Bernhard "bero" Rosenkränzer

bero@baylibre.com - bero@lindev.ch

Embedded Open Source Summit Europe 2023

Topics

- The current toolchain situation
- binutils
- compilers
- libcs
- C++ support
- Debuggers

The current situation

There is a well known, well documented (but slightly complicated) way to build crosscompilers -- some alternatives (some even good) have sprung up, but are not yet as widely known.

The traditional way

- Build binutils
- Build a minimal gcc crosscompiler (C only, no threads, ...)
- Build glibc
- Build gcc again with all needed features (including libstdc++ if you want C++ support)

Advantages of staying with this

- By far the most widespread -- most 3rd party libraries and applications have been tested in this setup, and chances are people on IRC, mailing lists, forums, ... have done this (and can likely help)
- Works well and optimizes well enough

binutils

- binutils is a collection of tools that deal with object files -- you might not have used them directly, your compiler uses them.
- There are 3 major implementations, and you're likely using all of them already:
  - GNU binutils (the "standard" implementation)
  - elfutils (shipped with almost all distributions to get libelf)
  - LLVM binutils (part of LLVM, built everywhere for Mesa etc.)

binutils

- elfutils provides good implementations of the tools it provides, but is lacking a linker -- you still need a linker, e.g. from GNU binutils, LLVM or MOLD.
- Tools in GNU binutils and LLVM binutils are pretty much interchangable and use mostly the same parameters
- Big advantage of LLVM tools: Crosscompilers built in. "`llvm-objdump --disassemble`" on an ARM binary works just fine on an x86 box (or vice versa) while you need per-architecture tools in GNU binutils

binutils - LLVM binutils

- In LLVM 10, the lld linker (up until recently, the main blocker for replacing binutils) has become good enough and compatible enough to replace ld.bfd and ld.gold for almost everything. And it's only getting better (while gold is seeing little attention).

binutils – mold linker

- The mold linker is a new linker started by the original developer of lld; it focuses on linking speed and parallelism, and is much faster than traditional linkers.
  Both clang LTO and gcc LTO supported
  Linker script support limited (and full support not planned)
  Works well for most applications/libraries

Compilers - gcc

- gcc has been the only serious compiler for a long time, and it still does a great job.
- Supports C, C++, Objective-C, Fortran, Ada, Go, D
- Supports x86 (32 and 64), ARM (32 and 64), PowerPC, MIPS, RISC-V, SPARC and many more architectures
- Generally supports latest versions of languages (C17, Most of C++20)
- Optimizes well

Alternative compilers: Clang

- There's another good compiler: clang (from LLVM).
- Supports C, C++, Objective-C, Fortran. Frontends for other languages are available out-of tree, some languages (Rust, Swift, Pony, ...) use LLVM
- Supports x86 (32 and 64), ARM (32 and 64), PowerPC, MIPS, RISC-V, SPARC and many more architectures, most notably AMD GPUs, NVIDIA PTX, WebAssembly, BPF
- Generally supports latest versions of languages (C17, C++20)
- Optimizes well, many sanitizers

Alternative compilers: Clang

- Clang is a crosscompiler by design: You don't have to build special per-target crosscompilers, clang can target any supported platform.
  Instead of having to build e.g. `aarch64-linux-gnu-gcc`, you can use `clang -target aarch64-linux-gnu` -- for any architecture.
- It's easier to get into Clang's code than into gcc's code.

Alternative compilers: Clang

- Many targets - including some GPUs - supported
- Performance of clang-built binaries is similar to gcc-built binaries. There are special cases where clang performs better, and others where gcc performs better. On average, their performance is similar with clang (16) carrying a slight advantage over gcc (13)
- Clang tries to be a drop-in replacement for gcc, implementing many gcc extensions
- Initial release was in 2012 (gcc: 1987)

Alternative compilers: Clang

- Apache 2.0 licensed (with both the advantages and drawbacks compared to gcc's GPL)
- There's a good chance you'll need LLVM anyway (it is used, among other things, by Mesa) -- but on the other hand, depending on some other decisions, you may need GCC anyway even if you opt for clang (libstdc++, libgcc_s)

Alternative compilers: Clang

- Compile time can be significantly shorter with clang especially when C++ is involved. Building LLVM with gcc takes almost twice as long as building it with clang.
- Modular code - most of LLVM/Clang functionality is contained in libraries, making it easy to create new programming languages, target new processors or architectures like WebAssembly, or embed functionality inside an IDE

Alternative compilers: Clang

- Fortunately, clang and gcc are binary compatible. You can link a gcc-built binary to a clang-built library and vice versa. In fact, you can even

```
gcc -O2 -o test1.o -c test1.c
clang -O2 -o test2.o -c test2.c
gcc [or clang] -o test test1.o test2.o
```

- If you want to mix compilers that way, you have to use gcc's support libraries (libgcc rather than compiler-rt) - clang can use gcc's, but not vice versa (at least without -nostdlib trickery)

Alternative compilers: TinyCC

- TinyCC is what its name implies - probably just about the smallest possible implementation of a full C99 compiler -- the compiler's source is smaller than 4 MB, and it takes mere seconds to compile.
  It has interesting uses (e.g. embedding inside an application), but doesn't optimize as strongly as clang or gcc.
  It is also limited to C (no C++). It might be interesting for small embedded devices.

Compilers: BSPs

- Many BSPs (Board Support Packages) that come with development boards contain a compiler.
  This compiler is usually a fork of an outdated version of gcc or clang (both of which, in the mean time, have typically added much better support for the hardware in question).
  Unless you're working on a very special device (not yet supported by the upstream compilers), it's usually good advice to ignore the BSP and build your own clang or gcc.
- Sometimes that means adding a few kernel patches to support newer toolchains - those patches are usually already written and relatively easy to find (try the kernel git repository).

Compilers: BSPs

- Special case for the Xtensa architecture and clang: Its support in upstream LLVM is limited (supported by LLVM libraries but not by clang, etc.)
- There is a vendor version that works well: https://github.com/espressif/llvm-project
- And of course the version we built for libapu, which is essentially the vendor branch rebased to LLVM 16.0.4 https://gitlab.baylibre.com/bero/llvm-for-libapu
- Hopefully both versions will go away with LLVM 17 merging everything that is needed.

Compilers: performance

Clang and gcc are similar in performance, across all architectures I’ve looked at (aarch64, riscv64, x86_64) – but there can be significant differences on some code.

Compilers: performance

`loop_unroll.cpp` from Adobe C++ Benchmarks on aarch64 operating on `int32_t`:

gcc: Total absolute time for int32_t do loop unrolling: 208.55 sec

clang: Total absolute time for int32_t for loop unrolling: 14.49 sec

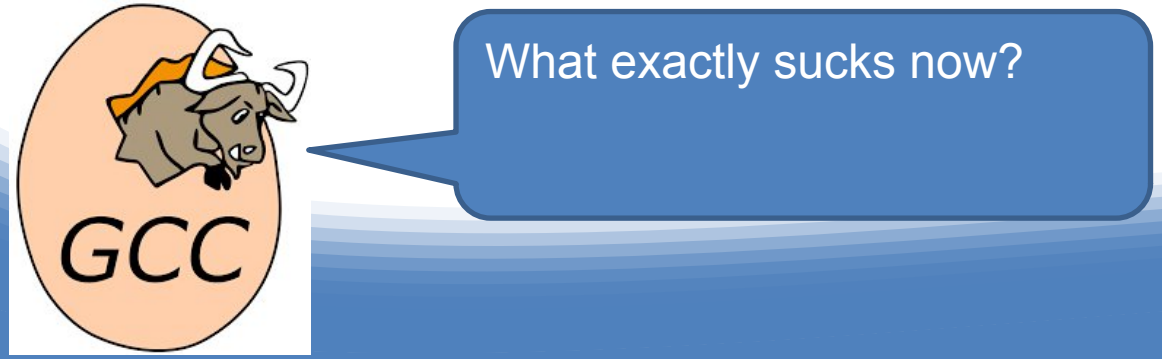
Compilers: performance

But we don’t even have to use a different benchmark suite to show the opposite...

`simple_types_constant_folding.cpp` on x86_64:

gcc: Total absolute time for float constant folding: 6.51 sec
Total absolute time for double constant folding: 7.13 sec

clang: Total absolute time for float constant folding: 8.33 sec
Total absolute time for double constant folding: 9.24 sec



Compilers: Conclusions

- gcc and clang are both good options. There is no clear winner.
- Both have been used to compile full systems (including the kernel). Most Linux distributions are built mostly with gcc, some (OpenMandriva, Android) and the BSDs are built mostly with clang. Some build-from-source distributions offer both choices.
  Yocto users can pull in the `meta-clang` layer to get clang support.

Compilers: Conclusions

- clang makes it easier (and, unless you're very familiar with gcc's code base, faster) to add new architectures and new languages, and is mostly built as a library. If you're planning to add architectures and new language, or to embed the compiler in your own projects, give clang a try.
- If you're using glibc, you need gcc to build it (for now, clang support in progress). If you don't need any of the extras offered by clang, you may want to go with gcc for everything.

Compilers: Multiple compilers are good!

- Try building your code with multiple compilers. Not only can it tell you which compiler works best for your particular use, it will also help find bugs – different compilers warn about different problems. Up until recently, only clang would warn about this:

```
void doSomething(char a[16]) {
   assert(sizeof(a) == 16);
}
```

libc: glibc

- For the system libc, glibc is the default option:
  - most widespread
  - most complete/most standards compliant
  - very well tested
  - most complete arch support (aarch64, arm, x86, x86_64, x32, RISC-V 64, alpha, C-Sky, hppa, ia64, m68k, microblaze, mips, powerpc, S/390, sh, SPARC)
- But:
  - code not very readable
  - compiles only with gcc (this is starting to change, but clang support is still experimental and doesn't work out of the box)
  - not very optimized for small systems
  - rather big (roughly 4 MB for ld.so, libc, libm, libpthread)

libc: musl

- Complete, fast and relatively small (785 kB)
- Designed for C11+ and POSIX 2008+, with many glibc, Linux and BSD extensions
- Supports aarch64, arm, x86, x86_64, x32, RISC-V 64, m68k, microblaze, mips, mips64, mipsn32, or1k, powerpc, powerpc64, s390x, sh
- Readable code
- Started 2011
- systemd pretends to need glibc, but happens to work with musl (at least the core components) with minor tweaks

libc: uClibc-ng

- Complete, fast and relatively small (1 MB in full config)
- Can be stripped down easily
- Focused on embedded systems
- Supports many processor types, including MMU-less: Aarch64, Alpha, ARC, ARM, AVR32, Blackfin, CRIS, C-Sky, C6X, FR-V, H8/300, HPPA, i386, IA64, LM32, M68K/Coldfire, Metag, Microblaze, MIPS, MIPS64, NDS32, NIOS2, OpenRISC, PowerPC, RISCV64, Sparc, Sparc64, SuperH, Tile, X86_64 and XTENSA

libc: klibc

- Written for the early bootup process, used in the initramfs of Debian and some derivates
- Subset of libc functions, optimized for size over performance
- More direct use of kernel structures avoids some type conversion (e.g. between different ideas of "struct stat")
- Extremely small (75 kB)
- But not powerful enough as a real world libc - might be an option for some embedded systems
- Uses GPL kernel headers, resulting license situation not 100% clear.

libc: LLVM libc (not yet complete)

- In its early stages, but some code is there.
- Potentially interesting in the future because:
  - Designed to work with sanitizers and fuzz testing from the start
  - Targeting C17 and up - not carrying around ancient cruft
  - Design goal: "Use source based implementations as far possible rather than assembly. Will try to fix the compiler rather than use assembly language workarounds."
  - The LLVM project has a track record of delivering good toolchain options

libc: bionic (Android)

- Originally based on the BSD libc, bionic is the libc used in Android.
- Currently supports ARM (32 and 64) and x86 (32 and 64)
- Rather well optimized because of vendor support for Android
- Used to be unusable for a regular Linux system - lacking e.g. SysV SHM needed for X11 - but has largely caught up
- Unfortunately, at the same time added some Android-isms that make it harder to use outside of a full Android system (APEX, system properties etc.), build system tied to the Android tree

libc: bionic (Android)

- Potentially makes it possible to use closed drivers written for Android in a regular Linux system without having to go through hacks like libhybris
- May be interesting to build Linux/Android hybrid systems

Other potential libc options

- newlib is limited to static linking - if you don't need dynamic linking, it may be for you.
  Most Zephyr builds today use newlib; but Zephyr is mostly transitioning to:
- picolibc is a fork of newlib and AVR libc. It frequently incorporates changes from newer newlib. Main differences:
  - New build system (meson)
  - Removed non BSD-licensed code
  - Handling of thread-local storage
  - stdio from AVR libc
  - Merged libc and libm
  - C++ exceptions support
  - Xtensa support

Other potential libc options

- dietlibc is optimized for small size and static linking - but not very actively maintained, and on something as low level as a libc, its GPL (not LGPL) license may be a problem if your system will allow building/installing/running custom code.
- BSD — Take the libc from your favorite BSD and port it to Linux (certainly doable — see Bionic)

libc: Conclusions

- There are many interesting options - for now:
- If you need **maximum compatibility** with other systems, go with **glibc**.
- If you need a full fledged, but **smaller** and **more memory efficient** libc, go with **musl**.
- If you need a subset of libc and want to **strip out unneeded components**, try **uClibc-ng**.
- If you want to experiment with **Android** features on regular Linux **Bionic** may be worth a try.

C++ support: libstdc++

- libstdc++ is part of gcc, used by almost all Linux distributions including some that use clang as their primary compiler (notable exception: Android)
- This is what almost everything is developed against - the easiest option if you don't want to tweak code to add missing `#include`s that happen to be ignored by libstdc++.

C++ support: libc++

- libc++ is an optional part of LLVM/Clang.
- It's newer and smaller than libstdc++, carries less cruft to support ancient code. Most benchmarks also show it performing better.
- Problem: You can't mix libstdc++ and libc++ (for obvious reasons, they provide the same standardized API and export the same symbols). You can't e.g. compile Qt against libc++ and expect a binary built with Qt/libstdc++ (such as pretty much any non-free software out there) to work.
- 3rd party applications (Chromium etc.) increasingly use libc++

C++ support: uClibc++

- uClibc++ is (was?) an attempt to write an STL implementation to go along with uClibc - a good idea (certainly you can strip out some parts of the STL when building an embedded system), but the last commit was in 2016. Don’t use it unless you’re prepared to revive the project.

C++ support: others

- Worth a mention because they are a fairly complete implementation of the STL of their time: STLport and Apache libstdcxx (last commit in both: 2008)
- Possibly worth a look because it is actively maintained and getting support for new C++ standards: MSVC's STL (has been opened up under the Apache license, but has not yet been ported to Linux and the project has no intention of doing so — but there may be some interesting platform independent code in there).

C++ support - conclusions

- If binary compatibility with other Linux distributions is a concern, go with **libstdc++**.
- If you're using clang and you care about performance and memory efficiency, try **libc++**.

Debuggers

- gdb has been the debugger to go to for a long time - initially released in 1986, and kept up to date (latest release: February 2023)
- More recently, lldb - a part of the LLVM project - has come along. Initially released in 2003, it has become a realistic replacement for gdb by now.
- Both tools do pretty much the same job, and both do it well.

Debuggers: GDB and LLDB

- LLDB provides many command aliases for gdb compatibility.
- LLDB's native syntax tends to be cleaner (designed 20 years later - less need to retrofit new features), but also more verbose
- LLDB has the edge in C++ support, and can evaluate expressions in the LLVM JIT
- Good news for remote debugger users: gdbserver and lldb-server speak the same protocol. You don't have to force users to use a specific debugger when deciding what (if any) debug server/stubs you put into a BSP/distro

Questions? Feedback? Bags of cash? ;)

- If you have any of the above, email me at bero@baylibre.com or bero@lindev.ch.

OpenMandriva Association