Building compiler runtimes “on demand”

Brooks Moses, Google LLC

2025 US LLVM Developers' Meeting

What is “on demand”?

In a typical C and C++ toolchain installation:

- Runtime libraries (`libclang_rt` , `libc++`, etc.) are pre-built when the toolchain is built.
- Built library files (`.so` and `.a`) are installed alongside the toolchain.

Runtimes built on demand:

- Runtime library sources are installed alongside the toolchain.
- Libraries are built by the toolchain when the user builds their target code.
    - Built library files are cached to provide acceptable build times.

This is a two-part talk:

- Part 1: Why this matters and how we do it.
- Part 2: How this affects LLVM runtime development

Part 1: Why “on demand” matters and how we do it.

Why would we do this?

Clang can support many architectures and ABIs.

- Different architectures (x86, Arm, etc.) and instruction set variants
- Different ABIs due to sanitizers, `libc++` configuration, etc.

Pre-compiled libraries support only one architecture and ABI

(or require complex tricks like glibc's `IFUNC` mechanism)

Toolchains may need many copies of the precompiled libraries, and complex logic to determine which one to use for a given set of compile options.

With "on demand" runtimes, we only need one copy.

Why would we do this?

Configurability for “embedded” platforms.

- Include or exclude parts of runtime libraries.
  - Coarse-grained: `libc` and `libc++` components (filesystem, threads, etc.)
  - Fine-grained: Does `printf` include floating-point formatting support?
- Compilation choices.
  - Optimize for size, or speed? Or both based on profile data?
- Even more microarchitecture and ABI variants.

Why would we do this?

Building runtimes on demand provides optimization opportunities.

- Link-time optimization (LTO) and inlining
  - Clang’s LTO requires IR bitcode inputs, not fully-compiled objects.
  - Runtime library functions are often small, so inlining is especially useful.
- Profile-guided optimization
  - Runtimes can be built with application-specific profile information.

Why would we do this?

Much better runtime-library development experience

- To test changes, just edit code and rebuild the target program.
- No need to fully re-build and re-install the precompiled runtime library copies.
- Avoids the “Did my change do nothing or did I forget to recompile?” problem.

How do we do this today?

The details are a bit specific to Google's Bazel-based build system.

- We have a "monorepo" and already build the explicit dependencies as part of building an application.
- Added build logic to add runtime libraries as an implicit "hidden" dependency to all C and C++ applications.
- Use a combination of LLVM "Bazel overlay" and custom build files.
- Currently building `libc++` and LLVM `libc` on demand.
  - Sanitizers and libclang_rt are future work.

Part 2: How this affects LLVM runtime development

How this affects LLVM runtime development

This isn’t the way runtime libraries have traditionally been deployed.

It requires new and different things from the libraries.

This section contains things I’d like LLVM developers to think about when creating LLVM’s runtime libraries.

Prefer simple build rules

The runtime libraries may need to be incorporated into the user's build system.

Tools like CMake and Bazel are very powerful.

> This means you can do very clever things with them.
>
> Cleverness is hard to translate.

Painful: [Insert your favorite overly-clever example here]
Better: Straightforward lists of files and build options

Prefer fewer build rules

Parsing build rules takes time.

A little extra time doesn't matter when doing a one-time toolchain build, but it will matter when it happens for every user build.

Painful: A separate build rule for every library function
Better: One or two build rules for the whole library

Painful: Build rules for tests in the same file as the rules for the library
Better: Tests are in a separate directory and makefile

Limit tools used at build time

Runtime libraries need to be built on the end-user's machine.

Every additional build tool becomes another dependency for users to install.

Painful: Users need tablegen to create runtime-library headers

Better: Runtime-library headers use `#ifdef` logic

Ideally, the only build tool needed is Clang itself.

Prefer self-contained source code

The LLVM project has about 50,000 source files. We don't want to install them all!

Runtime library sources should be self-contained as much as possible.

> This is relatively easy for things like `libc` and `libc++`.
>
> It gets more difficult for things like profiling instrumentation libraries, which may want to depend on data types from LLVM internals.

If a runtime library depends on parts of LLVM, the relevant parts should be separated into a minimal self-contained library.

Avoid generated header files

When libraries are pre-built, it doesn't matter if the headers are generated from templates.

For on-demand runtimes, generated headers are an annoyance:

- Generated code is an ephemeral build artifact.
- Debug info, assert failures, and error messages don't match the source files.

In my opinion, customizing the contents of header files with `#ifdef` logic usually provides a much better user experience.

Possibility: Modify the installed sources

The “installed” version of the runtime-library sources doesn’t need to be identical to the version in the source repository.

This provides some useful possibilities:

- We can generate headers with `tablegen`, if we do it at toolchain build time, and the result is target-agnostic.
- The installed makefiles don’t need to be the same as the repository makefiles. This is particularly useful when the repository makefiles include the rules to install the runtime-library sources.

Future work: Teach Clang to build runtimes?

Today, building the runtimes on-demand requires manually adding them to a project's build system dependencies.

Should we teach Clang to build them automatically?

> I.e., something like Rust's `cargo -Z build-std`.

If so, would Clang invoke `cmake`, or would it implement a small subset of CMake functionality so it can do this without an external separate tool?

Summary of things to think about

To enable “on-demand” building of runtimes, we want to be able to install a set of runtime library sources alongside the toolchain that are:

- Limited to only the necessary source files,
- Simple to build with arbitrary build systems, and
- Require a minimal set of additional build tools.

LLVM’s runtime libraries should be designed to enable this.

Other relevant talks

“LT-Uh-Oh: Adventures using LTO with libc,”, Paul Kirth and Daniel Thornburgh, 2:15 today, in this room.

“Climbing the ladder of complete: LLVM-libc past and future,” Michael Jones, yesterday. (Watch the recording!)

“Hand-In-Hand: LLVM-libc and libc++ code sharing,” Michael Jones and Christopher Di Bella, 2024 LLVM Developers Meeting.