Google

Understanding the LLVM build

LLVM Dev Mtg '23

Petr Hosek (phosek@google.com)

October 2023

I'm the technical lead for the LLVM toolchain team that supports lower level operating systems at Google.

In addition to supporting Fuchsia in LLVM, I maintain the CMake build, including the runtimes build.

I became involved in the build after Chris Bieneman's *Developing and Shipping LLVM and Clang with CMake* talk at 2016 LLVM Developers' Meeting.

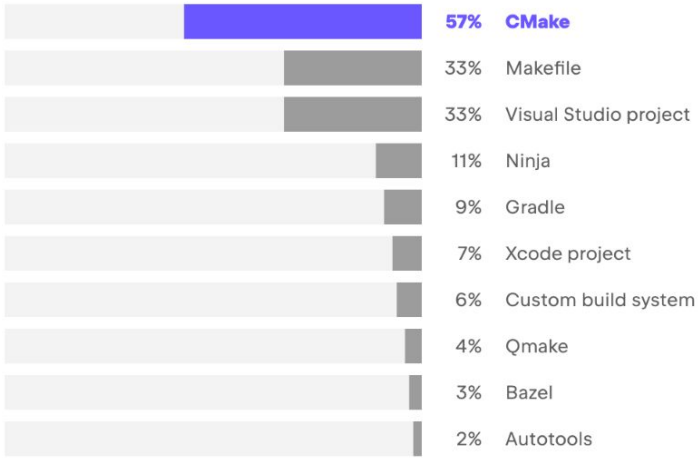
CMake manages the build process in an operating system and in a compiler-independent manner.

CMake is a cross-platform build-generator tool.

CMake does not build the project, it generates the files needed by your build tool (Ninja, Visual Studio, etc.)

Which project models or build systems do you regularly use?

Overall | Embedded | Games



I am on record as likening CMake to Stockholm syndrome for C++ engineers. It has become the de facto standard, for better or worse, as demonstrated by the clear lead it holds over its competitors.

Guy Davidson
Head of Engineering Practice, Creative Assembly

Agenda

01 Building projects & runtimes

02 Building toolchains

03 Building faster

04 Next steps

05 Q&A

01

Building projects & runtimes

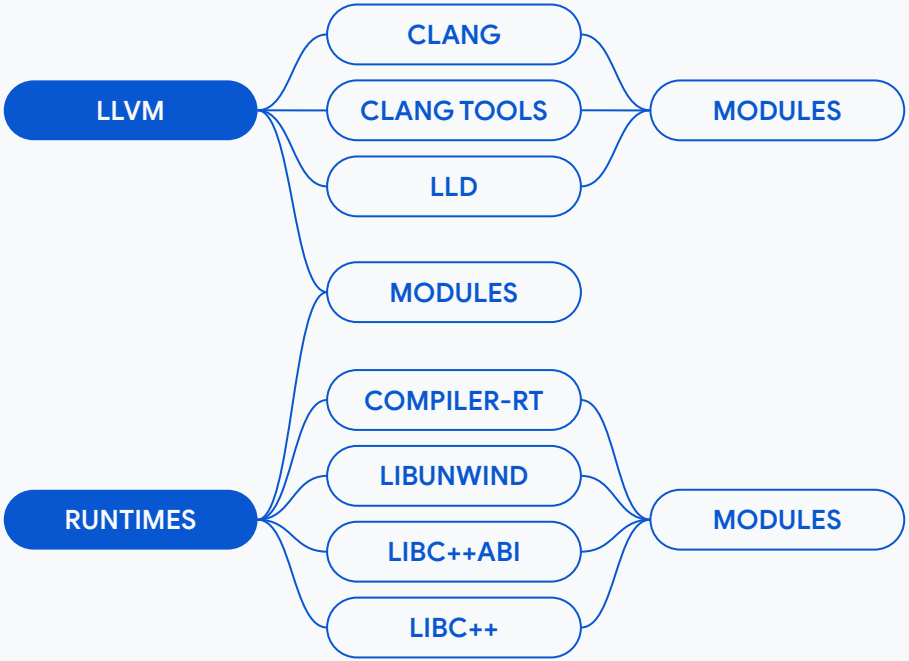
```
$ cmake -S llvm -B build -G Ninja -D CMAKE_BUILD_TYPE=Release
$ cmake --build build
$ cmake --build build --target check-all
$ cmake --install build
```

CMake stages

Configure	Generate	Build	Test	Install
Execute the CMake script `CMakeLists.txt` in the source directory.	Generate build targets for the selected generator in the build directory.	Build the project.	Run the project tests.	Install the appropriate header files, libraries, and executables.

```
$ cmake -S llvm -B build -G Ninja -D CMAKE_BUILD_TYPE=Release
$ cmake --build build
```

CMake source tree



```
$ cmake -S llvm -B build -G Ninja -D CMAKE_BUILD_TYPE=Release
$ cmake --build build
```

```
$ cmake -S llvm -B build -G Ninja -D CMAKE_BUILD_TYPE=Release
$ cmake --build build
```

```
$ cmake -S llvm -B build -G Ninja -D CMAKE_BUILD_TYPE=Release
$ ninja -C build
```

```
$ cmake -S llvm -B build -G Ninja -D CMAKE_BUILD_TYPE=Release
$ ninja -C build
$ ninja -C build check-all
$ ninja -C build install
```

```
$ cmake -S llvm -B build -G Ninja -D CMAKE_BUILD_TYPE=Release
$ ninja -C build
```

```
$ cmake -S llvm -B build -G Ninja \
    -D CMAKE_BUILD_TYPE=Release \
    -D LLVM_ENABLE_PROJECTS="clang;clang-tools-extra;lld" \
    -D LLVM_ENABLE_RUNTIMES="compiler-rt;libcxx;libcxxabi;libunwind"
$ ninja -C build
```

LLVM_ENABLE_PROJECTS

Controls which projects are built.

For example, you can work on Clang, Clang Tools or LLD by specifying
`-DLLVM_ENABLE_PROJECTS="clang;clang-tools-extra;lld"`

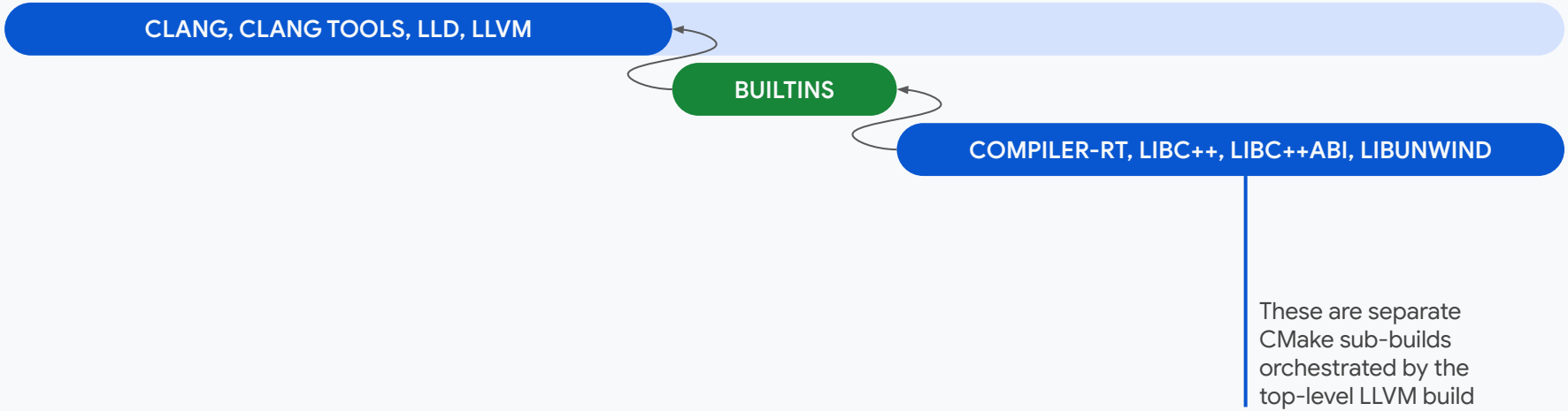
These are built using the *host compiler*.

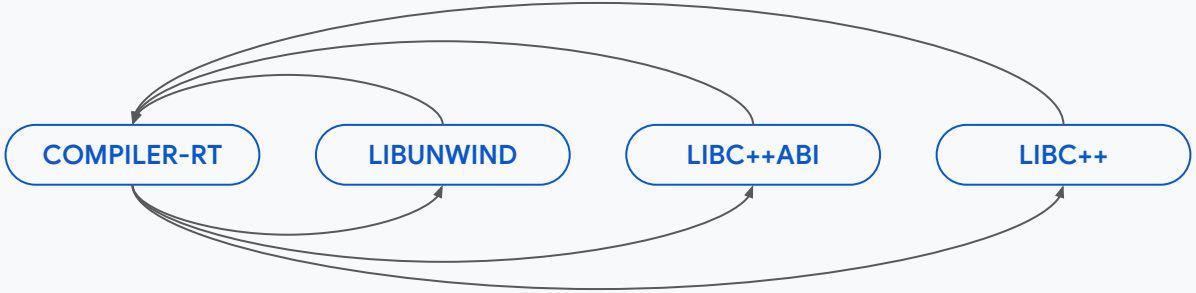
LLVM_ENABLE_RUNTIMES

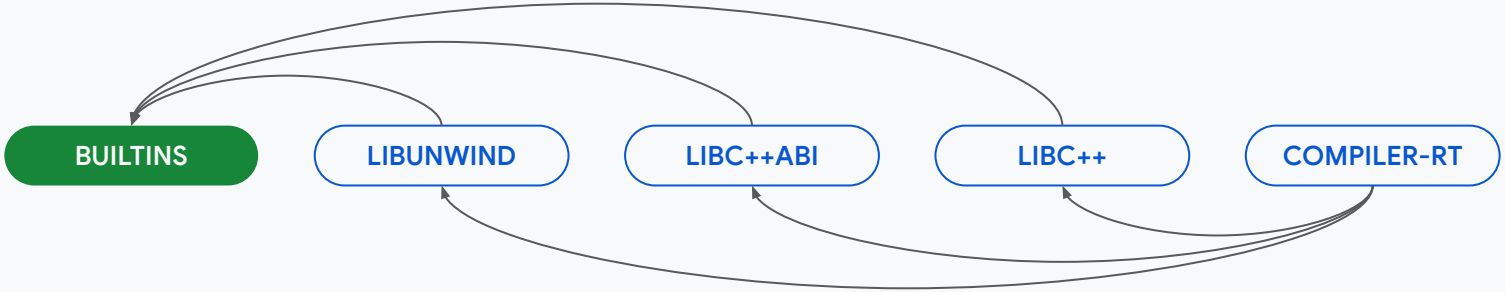
Controls which runtimes are built.

For example, you can work on compiler-rt, libc++, libc++abi or libunwind by specifying `-DLLVM_ENABLE_RUNTIMES="compiler-rt;libcxx;libcxxabi;libunwind"`

These are built using the *just-built compiler*.


CLANG, CLANG TOOLS, LLD, LLVM
BUILTINS
COMPILER-RT, LIBC++, LIBC++ABI, LIBUNWIND
These are separate CMake sub-builds orchestrated by the top-level LLVM build





Building runtimes

Runtimes build

- You use the LLVM build to drive the runtimes build by setting `-DLLVM_ENABLE_RUNTIMES="<name>;…"`
- Runtimes are built using the just built compiler as a sub-build.

Custom script

- You build LLVM and runtimes separately, typically using a custom script to drive individual sub-builds.
- This gives you a lot of flexibility, but also requires more maintenance.

```
$ cmake -S llvm -B build/llvm -G Ninja \
    -D CMAKE_BUILD_TYPE=Release \
    -D LLVM_ENABLE_PROJECTS="clang;clang-tools-extra;lld"
$ ninja -C build/llvm
```

```
$ cmake -S runtimes -B build/runtimes -G Ninja \
    -D CMAKE_BUILD_TYPE=Release \
    -D LLVM_ENABLE_RUNTIMES="compiler-rt;libcxx;libcxxabi"
$ ninja -C build/runtimes
```

```
$ cmake -S runtimes -B build/runtimes -G Ninja \
    -D CMAKE_BUILD_TYPE=Release \
    -D CMAKE_C_COMPILER="build/llvm/clang" \
    -D CMAKE_CXX_COMPILER="build/llvm/clang++" \
    -D LLVM_ENABLE_RUNTIMES "compiler-rt;libcxx;libcxxabi"
$ ninja -C build/runtimes
```

02

Building toolchains

LLVM_DISTRIBUTION_COMPONENTS

Controls which components to build and install.

All LLVM-based tools are components, as well as most of the libraries and runtimes.
Component names match the names of the build system targets.

```
$ cmake -S llvm -B build -G Ninja \
    -D CMAKE_BUILD_TYPE=Release \
    -D LLVM_ENABLE_PROJECTS="clang;clang-tools-extra;lld" \
    -D LLVM_DISTRIBUTION_COMPONENTS="clang;lld"
$ ninja -C build distribution
$ ninja -C build install-distribution
```

LLVM_INSTALL_TOOLCHAIN_ONLY

When enabled omits many of the LLVM development and testing tools as well as component libraries from the default `install` target.

Many of the LLVM tools are only intended for development and testing use and it is not recommended for distributions to include them.

```
$ cmake -S llvm -B build -G Ninja \
    -DCMAKE_BUILD_TYPE=Release \
    -DLLVM_ENABLE_PROJECTS="clang;clang-tools-extra;lld" \
    -DLLVM_DISTRIBUTION_COMPONENTS="clang;lld" \
    -DLLVM_INSTALL_TOOLCHAIN_ONLY=ON
$ ninja -C build distribution
$ ninja -C build install-distribution
```

CMake cache

The CMake cache may be thought of as a configuration file. The first time CMake is run on a project, it produces a `CMakeCache.txt` file.

CMake lets you initialize cache with a CMake script which executes before the root `CMakeLists.txt` file and has isolated scope.

See `clang/cmake/caches` for examples.

```
# MyCache.cmake
set(CMAKE_BUILD_TYPE Release CACHE STRING "")
set(LLVM_DISTRIBUTION_COMPONENTS "clang;lld" CACHE STRING "")
set(LLVM_ENABLE_PROJECTS "clang;lld" CACHE STRING "")
set(LLVM_INSTALL_TOOLCHAIN_ONLY ON CACHE BOOL "")
```

```
$ cmake -S llvm -B build -G Ninja -C MyCache.cmake
$ ninja -C build distribution
$ ninja -C build install-distribution
```

Multi-stage builds

Clang build supports bootstrap builds where build passes data from one stage into the next which enables complex multi-stage builds with a single CMake invocation.

This can be enabled by the `CLANG_ENABLE_BOOTSTRAP` option.

You can use `CLANG_BOOTSTRAP_PASSTHROUGH` to control which variables are passed through to the next stage in addition to the default set.

You can use `CLANG_BOOTSTRAP_TARGETS` to expose targets from later stages in the first stage build prefixed with `stage*-`.

```
$ cmake -S llvm -B build -G Ninja \
    -D CMAKE_BUILD_TYPE=Release \
    -D LLVM_ENABLE_PROJECTS="clang;clang-tools-extra;lld" \
    -D CLANG_ENABLE_BOOTSTRAP=ON \
    -D CLANG_BOOTSTRAP_PASSTHROUGH="CMAKE_INSTALL_PREFIX" \
    -D CLANG_BOOTSTRAP_TARGETS="check-all"
$ ninja -C build stage2
$ ninja -C build stage2-check-all
$ ninja -C build stage2-install
```

Cross-compiling runtimes

Runtimes build supports cross-compiling runtimes for multiple targets.

Use `LLVM_BUILTIN_TARGETS` to targets for compiler-rt builtins. To pass a per-target variable to the builtins build, you can set `BUILTINS_<target>_<variable>`.

Use `LLVM_RUNTIME_TARGETS` to targets for compiler-rt builtins. To pass a per-target variable to the builtins build, you can set `RUNTIMES_<target>_<variable>`.

The build targets are available as `builtins-<target>` and `runtimes-<target>`.

```
# MyCache.cmake
set(LLVM_BUILTIN_TARGETS "x86_64-linux-gnu;aarch64-linux-gnu" CACHE STRING "")

foreach(target ${LLVM_BUILTIN_TARGETS})
  set(BUILTINS_${target}_CMAKE_SYSTEM_NAME "Linux" CACHE STRING "")
  set(BUILTINS_${target}_CMAKE_SYSROOT "${SYSROOT_${target}}" CACHE PATH "")
  …
endforeach()
```

```
# MyCache.cmake
set(LLVM_RUNTIME_TARGETS "x86_64-linux-gnu;aarch64-linux-gnu" CACHE STRING "")

foreach(target ${LLVM_RUNTIME_TARGETS})
  set(RUNTIMES_${target}_CMAKE_SYSTEM_NAME "Linux" CACHE STRING "")
  set(RUNTIMES_${target}_CMAKE_SYSROOT "${SYSROOT_${target}}" CACHE PATH "")
  …
endforeach()
```

```
$ cmake -S llvm -B build -G Ninja -C MyCache.cmake \
    -D SYSROOT_x86_64-linux-gnu=<path> \
    -D SYSROOT_aarch64-linux-gnu=<path> \
$ ninja -C build runtimes-x86_64-linux-gnu runtimes-aarch64-linux-gnu
```

Multilibs

Runtimes build also has basic multilib support.

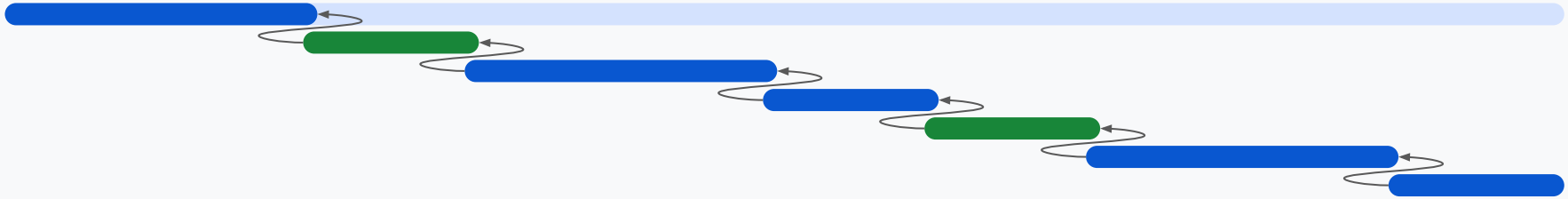
Use `LLVM_RUNTIME_MULTILIBS` to define your multilib variants and map those to targets with `LLVM_RUNTIME_MULTILIB_<multilib>_TARGETS`.

Use `RUNTIMES_<target>+<variant>_<variable>` to pass a variable to the variant build. The build targets are available as `runtimes-<target>+<variant>`.

Note that multilib support currently *does not* include compiler-rt—which includes builtins—this limitation might be lifted in the future.

```
# MyCache.cmake
set(LLVM_RUNTIME_MULTILIBS "asan" CACHE STRING "")
set(LLVM_RUNTIME_MULTILIB_asan_TARGETS
    "x86_64-linux-gnu;aarch64-linux-gnu" CACHE STRING "")

foreach(target ${targets})
  set(RUNTIMES_${target}+asan_LLVM_USE_SANITIZER "Address" CACHE PATH "")
  …
endforeach()
```



03

Building faster

Use release mode

Debug build tends to be slower and produce large amount of debug info.

When you do not intend to use the debugger, release mode with assertions is a more efficient alternative; this can be enabled by `-DLLVM_ENABLE_ASSERTIONS=ON`.

Use faster tools

Whenever possible, use faster tools and libraries:

- `LLVM_OPTIMIZED_TABLEGEN`
- `LLVM_ENABLE_LLD`
- `LLVM_ENABLE_LIBCXX`

LTO+PGO+BOLT optimized host compiler can speed up your build by 25-50%.

We have an examples of LTO, PGO and BOLT builds in `clang/cmake/caches`.

Build less code

Only build what you need:

- `LLVM_TARGETS_TO_BUILD=Native`
- `LLVM_ENABLE_PROJECTS=clang;…`
- `LLVM_ENABLE_RUNTIMES=libcxx;…`
- `LLVM_DISTRIBUTION_COMPONENTS=llvm-ar;…`

Leverage caching

Tools like ccache and distcc *can* speed up builds.

To use these with CMake, you can set these as compiler launchers
`-DCMAKE_{C,CXX}_COMPILER_LAUNCHER={ccache,distcc,…}`

04

Next steps

Build all runtimes in the runtimes build

Some projects like compiler-rt or libc can be built as standalone, as part of LLVM, or as part of the runtimes build which complicates maintenance.

Supporting only the runtimes build would let us remove a lot of the complexity.

This may require improvements to the runtimes build to support all existing use cases.

Build builtins with other runtimes

We should be able to build builtins together with other runtimes in a single build by carefully ordering feature checks and setting the dependencies correctly.

This would simplify and speed up the runtimes build, but might require significant refactoring.

Use modern CMake

Many of the concepts in our build date back to CMake 2.8 (released in 2009), but CMake keeps on evolving (current is 3.27, LLVM requires at least 3.20).

We should be following Modern CMake principles—this would simplify our build and make it more approachable to newcomers.

We should be treating build like code; build keeps on evolving to accommodate new use cases, and we need to dedicate resources to refactorings and cleanups to avoid too much technical debt.

05

Q&A

Whoever makes a problem visible gets blamed for its existence...

Build systems make *many* problems visible.

Jussi Pakkanen
Behind the Scenes of a C++ Build System - CppCon 2019