LLVM Toolchains in Nixpkgs

LLVM Distributors Conference
2021

JOHN.ERICSON
@OBSIDIAN.SYSTEMS

LUKAS EPPLE
STERNENSEEMANN@SYSTEMLI.ORG

Nix Ecosystem Basics

Layering

Layering — bird’s eye view

Nixpkgs — Plans

- Interesting
- Complex
- Cheap

Nix — Executes

- Boring
- Simple
- Expensive

Not unlike, say, CMake and Ninja, at first glance

Layering — mole's snout view

Nix — *sandboxed* dumb builder

- Build artifacts not just files
  - Directories or files
  - Tracks references to other build artifacts
- Don't choose where artifacts "go"
  - All build steps produce 1 or more *new* `/nix/store/<hash>-name`
- "Hermetic"
  - Only see your inputs' chosen outputs
  - And their references
- No "global" views
  - Keep that in mind for later

Nixpkgs — package *all* the things

- Big functional program
  - Evaluates to Nix dependency graph
  - Usual granularity is Build step = entire package
- Self-contained
  - Linux: auto-download bootstrap binaries
  - macOS: only few exceptions
    - Some non-redistributable libraries "peeked" at
    - Still, no need to install XCode, etc.
- The abstractions *and* the packages that use them
  - Iterate faster than waiting for new CMake!

Cross compilation in Nixpkgs

Platforms we've targeted

- OSes:
  - Linux
  - Darwin
  - NetBSD
  - Windows (MinGW)
- Phones:
  - iOS
  - Android
- Arches:
  - x86
  - ARM
  - RISC-V
  - Power
  - AVR
  - VC4
  - WASM
  - JavaScript for Haskell

- Libc's
  - glibc for Linux
  - Musl for Linux
  - newlib for freestanding
  - WASI for WASM
  - (Default for Darwin and NetBSD)

Can always *attempt* building any package, obviously can't get much farther than toolchain itself for tiny embedded

Relevant code in Nixpkgs:

- lib/systems/examples.nix
- lib/systems/parse.nix

How does it work?

Quick GNU Platform name recap

- Build platform
  - Where the package is built
  - Just an implementation detail
    - Whew, not part of dependency interface!
- Host Platform
  - Where the package runs
  - Part of interface
- Target Platform
  - Where the package *produced* by this package runs
  - Part of interface
  - Bad
    - Libraries don’t care
    - Compilers can and should simultaneously target multiple platforms
  - LLVM being multi-target frees us of this!

How it all works — Two views

- Unresolved package
  - Abstract, especially dependencies
  - Compositional
  - Contained in many (possible) plans
- Entire plan
  - Everything concrete
  - Rapidly prototype
  - Contains my packages

Too many kinds of dependencies?

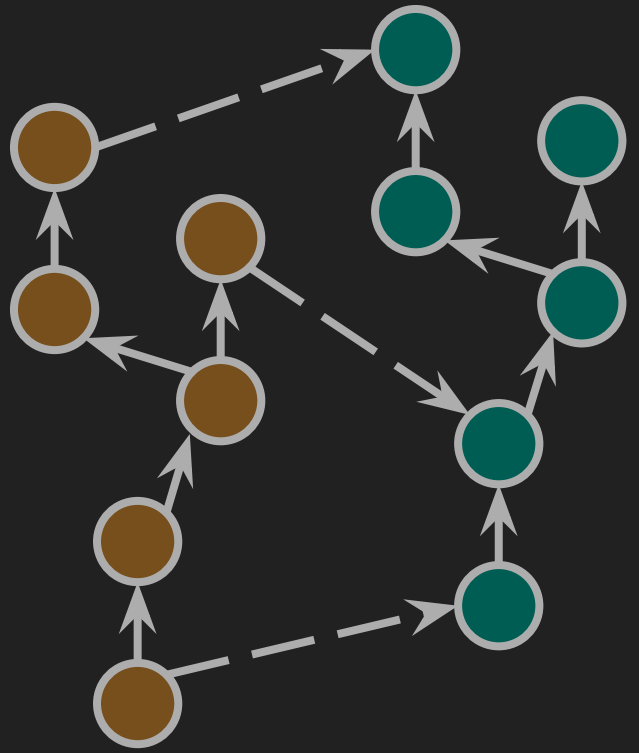
- Yesterday: Library vs Executable
- Yesterday: Build-time vs run-time
- Today: Cross adds multiple platforms
  - ...how?

Key insight: when need ⇒ what platform

- Build dependency chain through different platforms
  - Build, Host, and (if applicable) Target should be thought *relative* current package
- Observations
  - “My build platform is my build tool’s host platform”
  - “My host platform is my build tool’s target platform”
  - “I don’t care about my build tools’ build platform”
  - “I don’t care about my children’s target platform”
- *Local* choice of dependencies determines bootstrapping
  - Very little “global planning needed”
  - Only exception: toolchain wrappers because they agglomerate deps

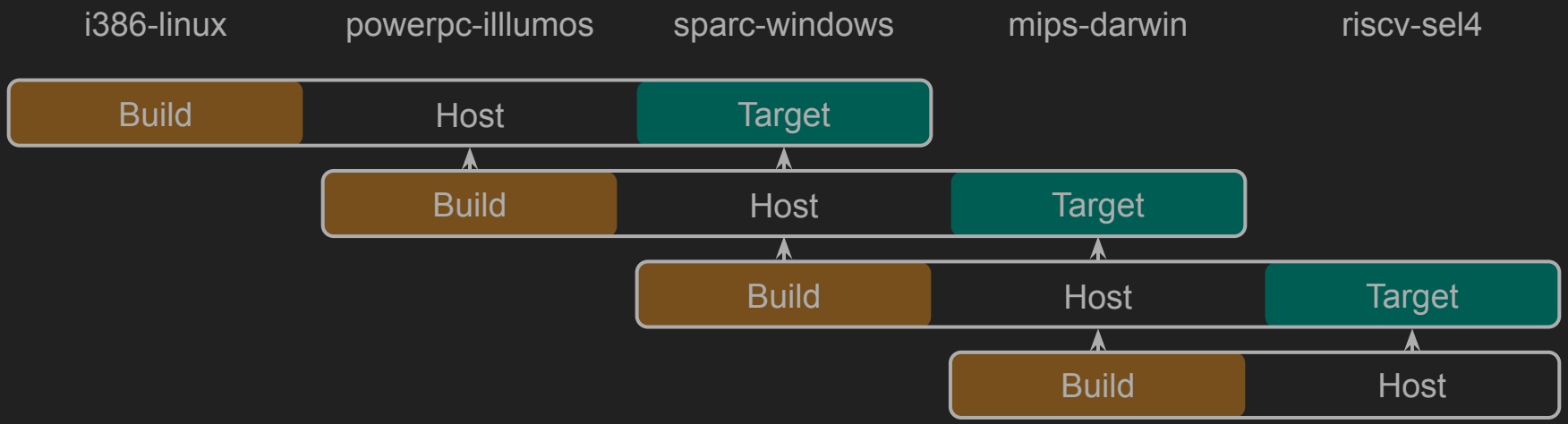
Entire plan — package sets per stage

- Run-time deps are resolved to the current stage
- Build-time deps are resolved to the previous stage



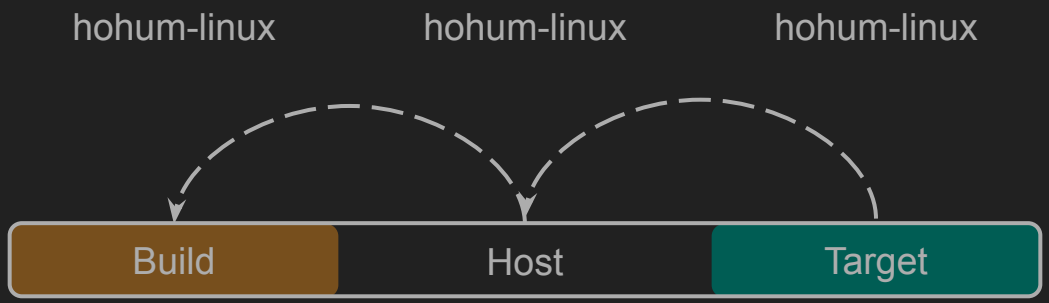
Platforms and stages: “Sliding Window Principle”

- Given chain of platforms
- Get chain of stages each building the next



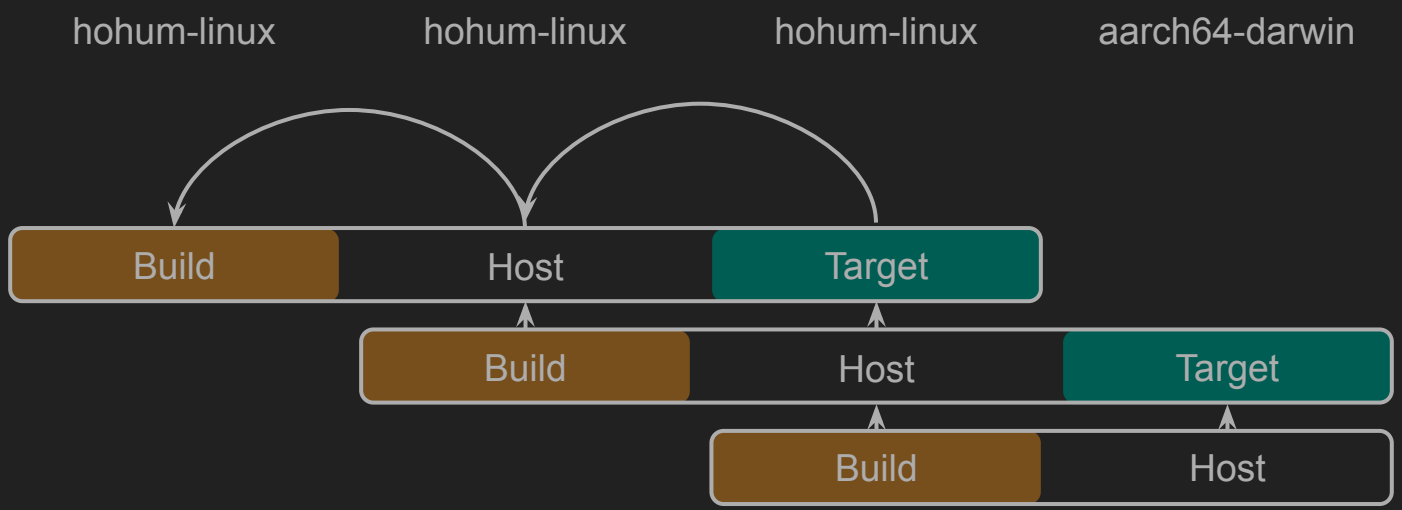
Putting it together: Native

- Fixed point further constraints platforms



Putting it together: Cross

- Need 2 extra stages: tools and final deployed packages



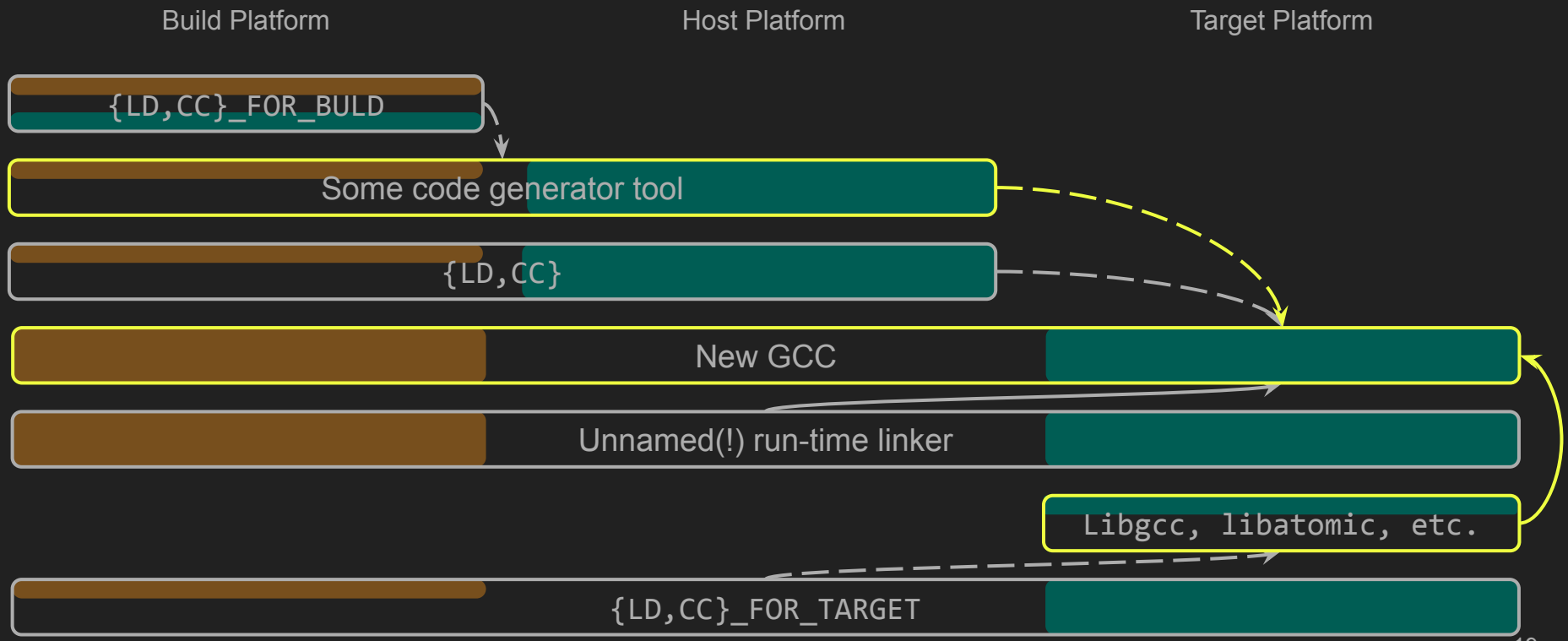
General Principle — Be Parametric!

- Elimination is bad
  - More code/branches to debug and reason about
  - Cross compilation users rare, need to bandwagon on better tested code!
  - Maxim: Write code with empathy for a symbolic/abstract interpreter!
- Native should just be “identity cross”
  - Like logic by default build = host is unknown
  - Native is the opt-*in* to it being true
- isCross Predicate is risky!
  - Vague: Build != Host, Host != Target?
- Shout-out to Exherbo
  - Gentoo fork
  - Pioneered this approach
  - https://exherbo.org/docs/multiarch-pr.html

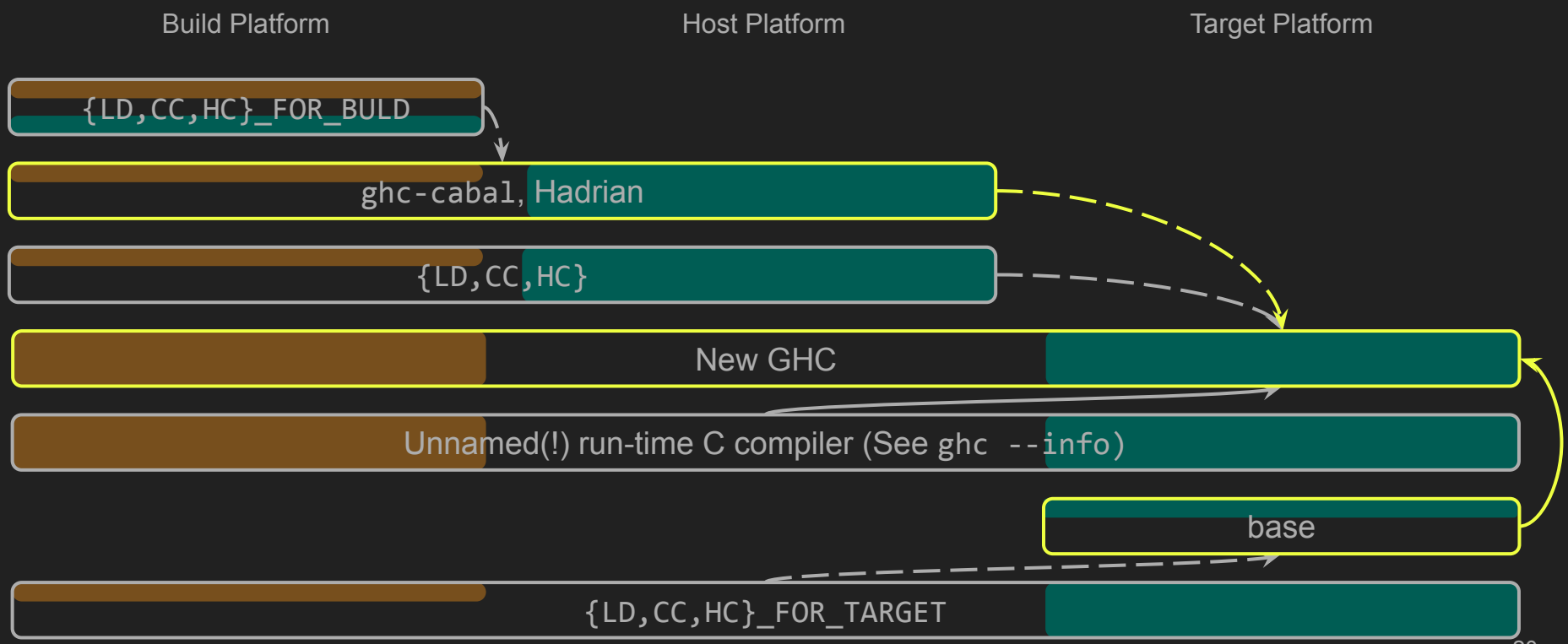
Problem: Compilers & stdlibs built together

- “Canadian cross” (Build != Host != Target) most illustrative…
- Library built on Build, runs on Target
  - Host is leapfrogged!
- Bootstrapping stages actually “braided” rather than linearly “chained”

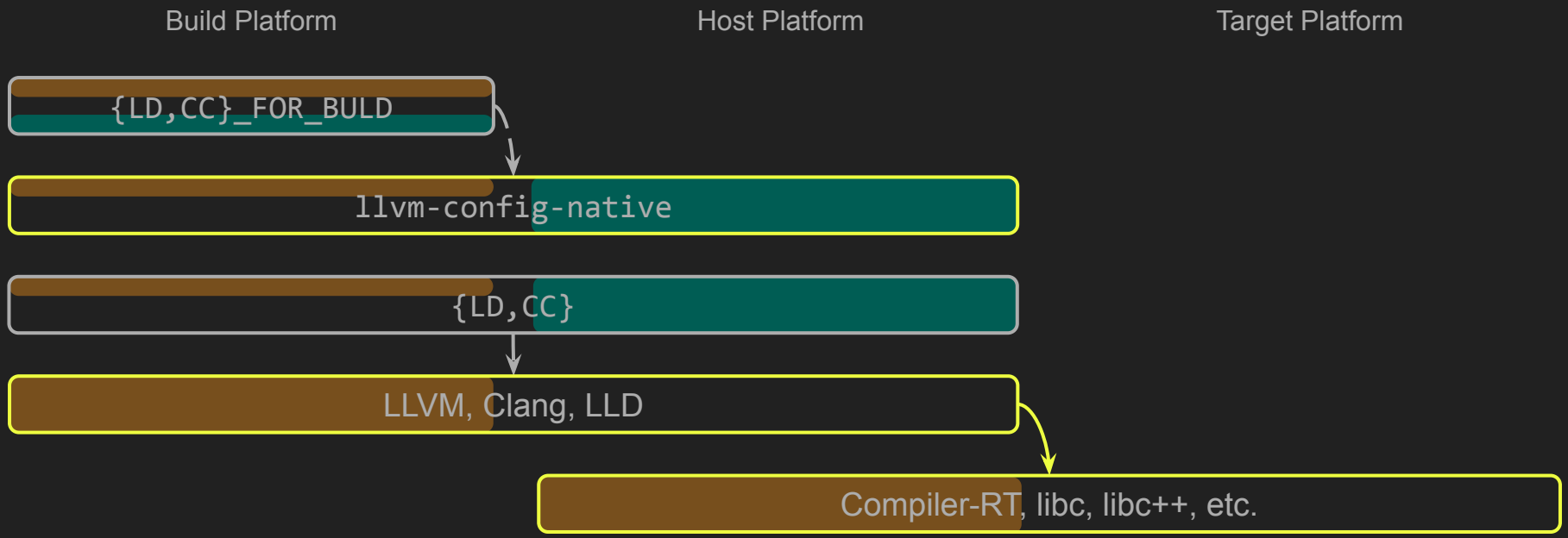
“Canadian cross” mess: GCC



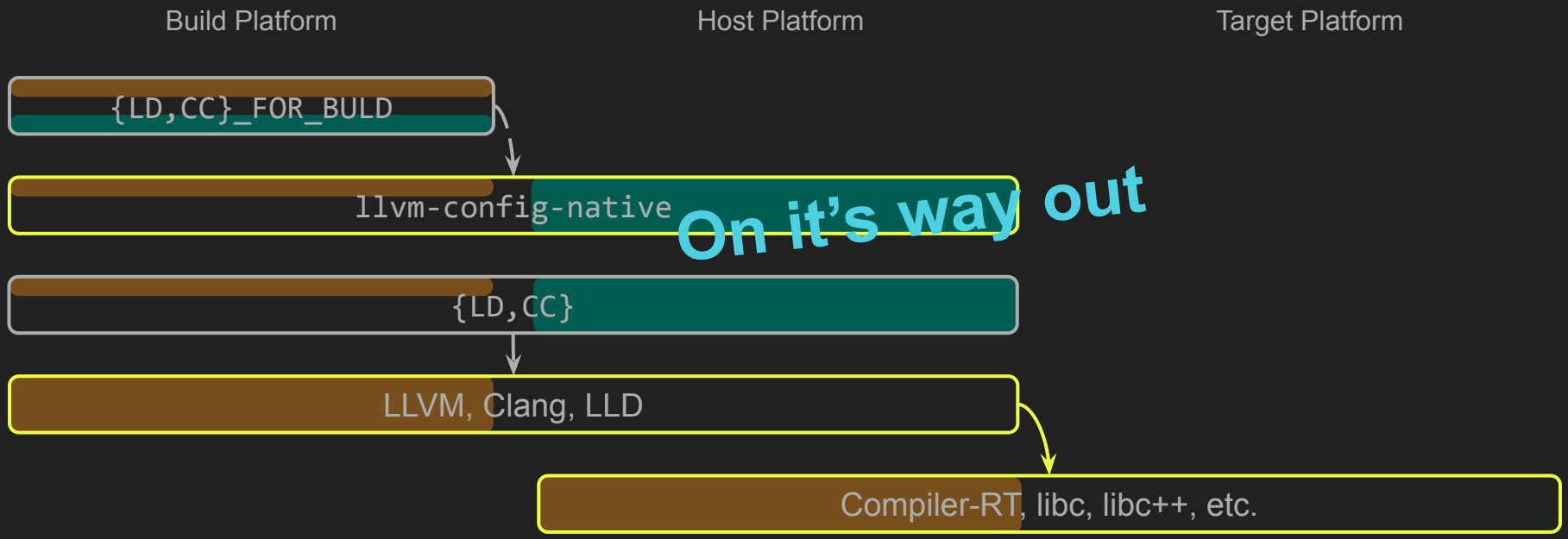
“Canadian cross” mess: GHC



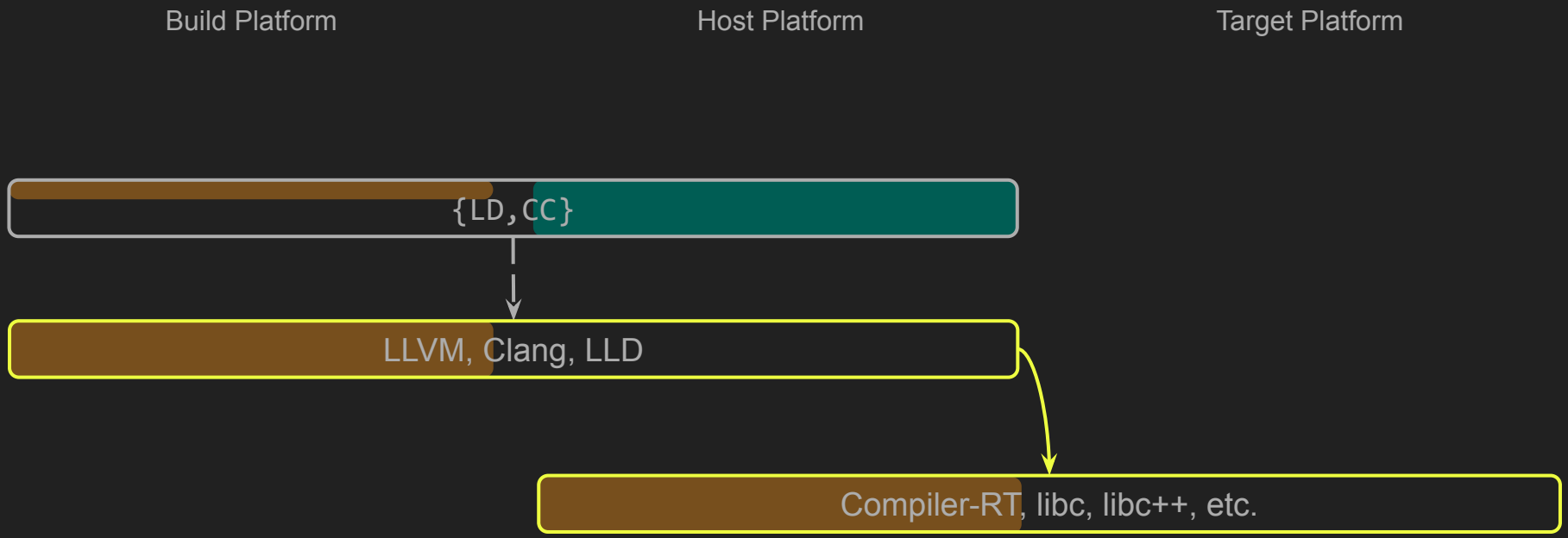
“Canadian cross” *non*-mess: LLVM



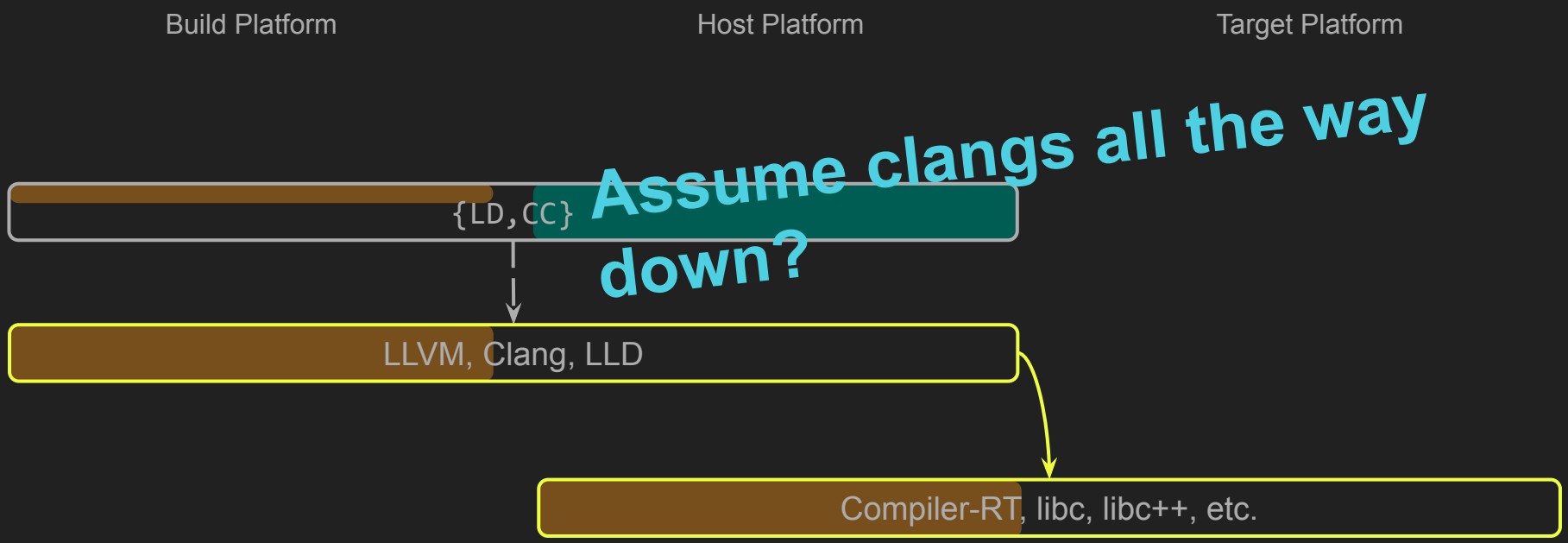
“Canadian cross” *non*-mess: LLVM



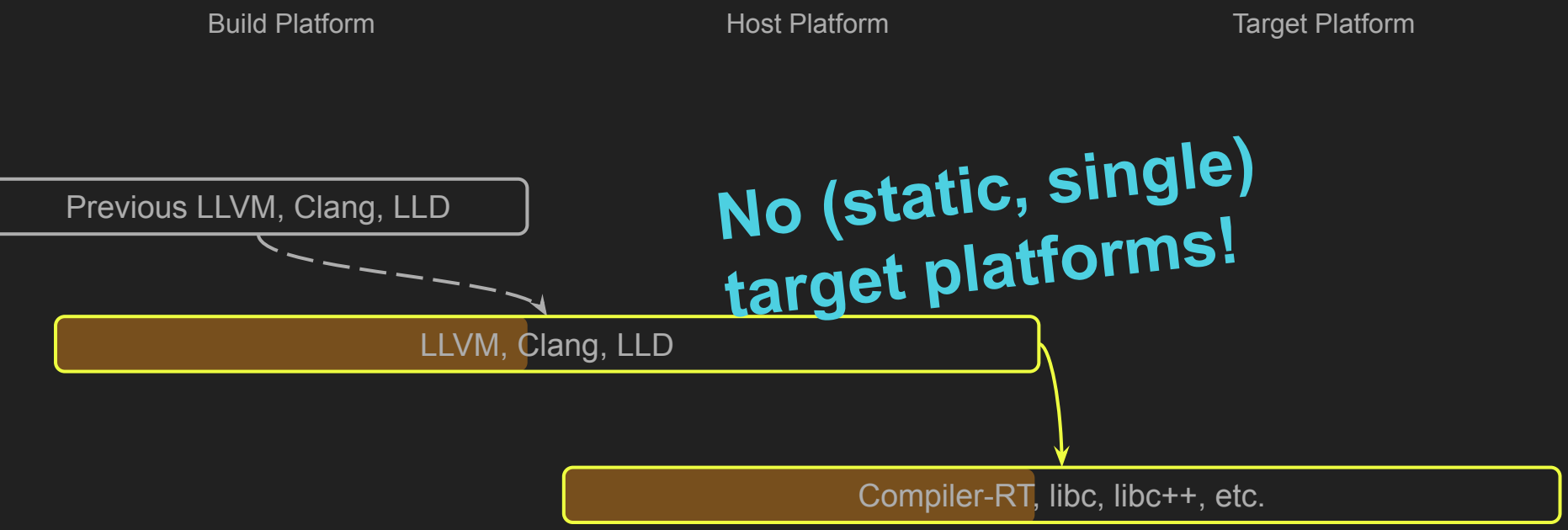
“Canadian cross” *non*-mess: LLVM



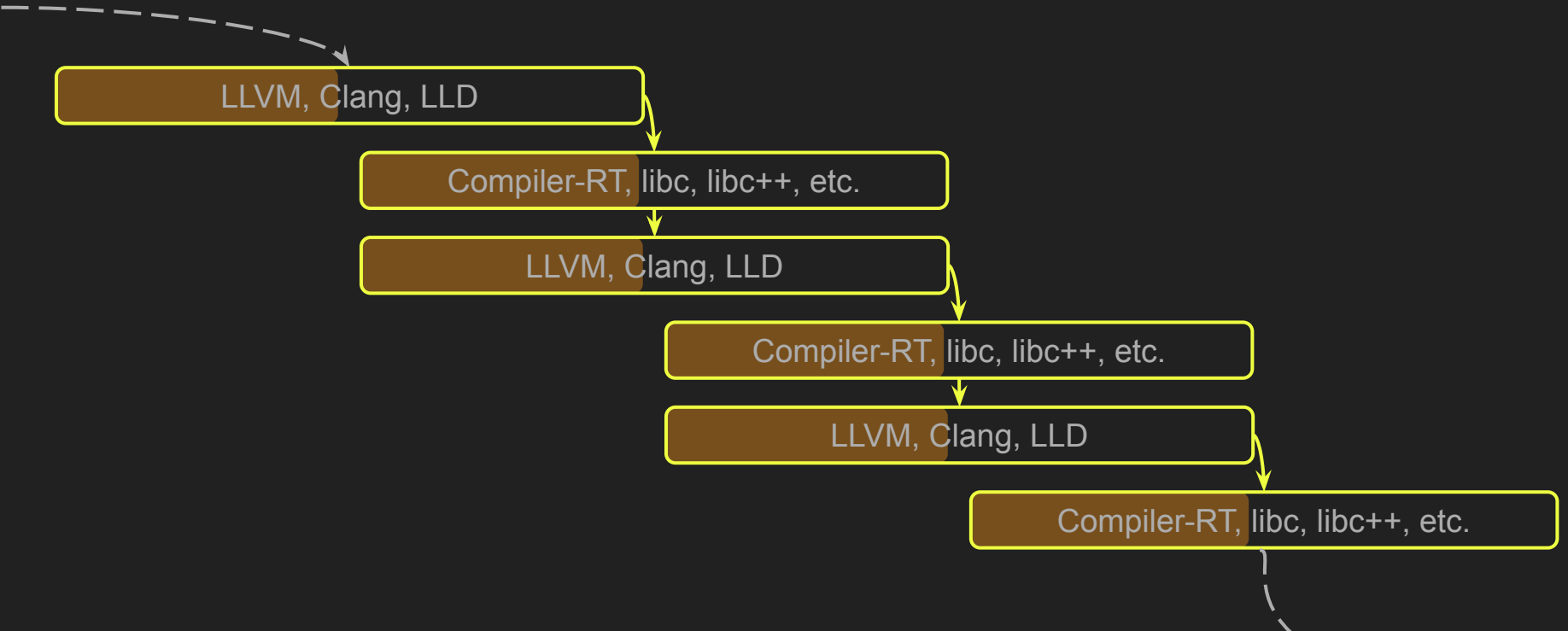
“Canadian cross” *non*-mess: LLVM



“Canadian cross” *non*-mess: LLVM



Dragons all the way down!



Concrete code

- “Wrappers” (put together toolchains)
  - CC: pkgs/build-support/cc-wrapper/default.nix
  - “bintools”: pkgs/build-support/bintools-wrapper/default.nix
    - Made up name trying to find “non-branded” binutils
- LLVM:
  - pkgs/development/compilers/llvm/{5..13,git} — directories for each version
  - …/llvm/13/default.nix — mini “package set” and bootstrapping
  - …/llvm/13/compiler-rt/default.nix — Compiler-RT
  - …/llvm/13/llvm/default.nix — LLVM
  - …/llvm/13/clang/default.nix — Clang
  - …/13/libcxx/default.nix — libc++
  - …/13/libcxxabi/default.nix — libc++abi
  - …

Learned Principles

- Packages shouldn't manage their own bootstrapping, the distro should.
  - Compilers and run-times should be built separately, so every package installs artifacts just for one platform.
  - Otherwise we are back to GCC, GHC complexity
- Make cross compilation the default, native compilation should "just" be cross compilation such that "build = host".
  - OK to assume can run “ambient” tools, don’t assume can link ambient libs!
- Runtime libraries should be "unspecial" --- like any other library --- at least one building.

LLVM very close on all them!

Requests for LLVM

Very happy overall, but doesn’t mean things are perfect!

1. GNUInstallDirs

- Nixpkgs typically splits libraries into regular and “dev” built artifacts
- Typically goal, but unusual means
- We do
  - Code: `outputs = [ “out” “dev” ];`
  - Becomes during build and after:
    - `/nix/store/<hash>-<libname>` (out is special-cased convention default)
    - `/nix/store/<hash>-<libname>-dev`
  - CMake gets:
    - -DCMAKE_INSTALL_LIBDIR=$out/lib
    - -DCMAKE_INSTALL_INCLUDEDIR=$dev/include
    - …
  - Note we are passing *absolute* paths — odd but allows!

1. GNUInstallDirs — Why we want?

- Avoid headers at runtime of course
  - but that's boring, space is cheap
- Stress-test installations in more different directories
  - Our non-FHS is very unstandard, unlikely to be tested upstream
  - But this does make the *possibility* of such things clear upstream
  - Want to avoid e.g. Clang fishing things out with brittle exe-relative paths (e. g. LLVMgold.so)
- Upstreaming our patch in progress:
  - Current diffs: D100810, D99484
  - Thanks to reviews so far, including Petr & Saleem talking today!

2. -B and -resource-dir

- With so many separate installs, wrapper passes lots of these
- On the other hand, have no need for `lib/<target>/include` type “subtrees”
- Reverse engineered a bit painstakingly
- Why not one way for all these purposes?
  - Or spicier, just use -L, -I, -isystem, etc. and trust the user not to shoot themselves in foot?

3. Clarify compiler-rt ⇔ libc dependencies

- Libc need builtins
- Sanitizers need libc
- Builtins fallbacks may need libc (?) (atomics?)
- ???

3. Clarify compiler-rt ⇔ libc dependencies

Code sample (a3a462734aa18ee1cdecf0c9d85bdca0503f3146:pkgs/development/compilers/llvm/13/compiler-rt/default.nix#L26-L52):

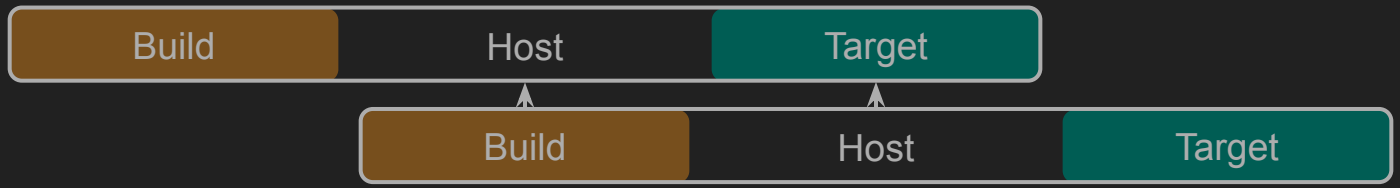
```
cmakeFlags = [
  "-DCOMPILER_RT_DEFAULT_TARGET_ONLY=ON"
  "-DCMAKE_C_COMPILER_TARGET=${stdenv.hostPlatform.config}"
  "-DCMAKE_ASM_COMPILER_TARGET=${stdenv.hostPlatform.config}"
] ++ lib.optionals (useLLVM || bareMetal || isMusl) [
  "-DCOMPILER_RT_BUILD_SANITIZERS=OFF"
  "-DCOMPILER_RT_BUILD_XRAY=OFF"
  "-DCOMPILER_RT_BUILD_LIBFUZZER=OFF"
  "-DCOMPILER_RT_BUILD_PROFILE=OFF"
] ++ lib.optionals ((useLLVM || bareMetal) && !haveLibc) [
  "-DCMAKE_C_COMPILER_WORKS=ON"
  "-DCMAKE_CXX_COMPILER_WORKS=ON"
  "-DCOMPILER_RT_BAREMETAL_BUILD=ON"
  "-DCMAKE_SIZEOF_VOID_P=${toString (stdenv.hostPlatform.parsed.cpu.bits / 8)}"
] ++ lib.optionals (useLLVM && !haveLibc) [
  "-DCMAKE_C_FLAGS=-nodefaultlibs"
] ++ lib.optionals (useLLVM) [
  "-DCOMPILER_RT_BUILD_BUILTINS=ON"
  # https://stackoverflow.com/questions/53633705/cmake-the-c-compiler-is-not-able-to-compile-a-simple-test-program
  "-DCMAKE_TRY_COMPILE_TARGET_TYPE=STATIC_LIBRARY"
] ++ lib.optionals (bareMetal) [
  "-DCOMPILER_RT_OS_DIR=baremetal"
] ++ lib.optionals (stdenv.hostPlatform.isDarwin) [
  "-DDARWIN_macosx_OVERRIDE_SDK_VERSION=ON"
  "-DDARWIN_osx_ARCHS=${stdenv.hostPlatform.darwinArch}"
  "-DDARWIN_osx_BUILTIN_ARCHS=${stdenv.hostPlatform.darwinArch}"
];
```

3. Clarify compiler-rt ⇔ libc dependencies

- We do (usually)
  - “Bare metal” style compiler-rt
  - Libc
  - (optional) fuller compiler-rt for sanitizers
- Idea: Split library instead of lots of conditionals?
  - Fine grained dependencies mean less combinatorial explosion to maintain
    - Conditions — arbitrary boolean expressions — SAT
    - Dependencies — horn clauses — HORNSAT
- C.F. Rust’s `core` vs `std`
  - Nice that core is (basically) the same whether freestanding or hosted
  - Good for code reuse too!

4. Use of "target" in source code interfaces

Per “sliding window” from earlier…



- Clang’s *target* is code being compiled’s *host*
- D44753
  - My first LLVM diff, years back
  - Worried about “target” in `__is_target_arch` and friends, when actually means host
    - If compiling non-compiler, not so bad
    - But If compiling legacy compiler with hard-coded target, can be confusing
      - Two different contradictory uses of “target”!
  - I wasn’t clear then, too late to change anyways, but hopefully clearer now :)

New things we are doing

pkgsLLVM (since May 2021)

- Package sets with altered toolchain settings:
  pkgsMusl, pkgsStatic, pkgsCross.x86_64-netbsd, …
- pkgsLLVM: “natively” cross-compiled, Clang, LLD, Compiler-RT, …
- The Good: benefit from interest in cross, upstream clang support
- The Bad: glibc, libgcc, … → use GCC for the moment
- The Ugly: boost, ...
  - build systems with poor cross support
  - friction from differences between nixpkgs’ cc-wrapper and vanilla Clang
- Composable: pkgsMusl.pkgsLLVM
- Not all combinations work: pkgsLLVM.pkgsMusl
- Future challenges: PGO, LTO, building your entire system with LLVM?

Simplifications

- Would like to simplify horrendous toolchain wrappers
- But GCC being unlike LLVM prevents this
- Solution: nixpkgs/#132343 repackage GCC per component too!
- Then have cake and eat it too:
  - Split components
  - Do the same things for both compilers
    - Less elimination, per earlier maxim

New platforms

- nixpkgs/#72366 LLVM Windows following @mstorsjo's work
  - MinGW and official SDK / MSVC libs/headers
- nixpkgs/#82131 FreeBSD, following our NetBSD support
  - Eventually, NixOS/k*BSD ?!?!
- Redox-OS
  - Some support thanks to @aaronjanse, merged Nixpkgs + https://github.com/nix-community/redoxpkgs
  - Need to replace some pre-built binaries
- Less pre-built Android?
  - nixpkgs/#117591 started, thanks to @s1341
- Fuschia?!
- Maybe after this talk we can stop reverse-engineering, start collaborating! :)

That’s all, thanks!

We want to build all the the things, help us help you!