Building with a custom toolchain

Guillaume Massé edited this page on 20 Mar 2017 · 10 revisions

Introduction

This page explains how to use a custom prebuilt C/C++ toolchain contained in an external repository. The example will be using `new_http_repository` because it's a common use case, but the procedure for `new_local_repository` and all the other types is similar. This example will be using precompiled GCC 4.9 from Linaro along with Clang as the C/C++ compiler, but other toolchains should be similar.

From experience getting several toolchains to work both with Bazel and other build systems, each one is different. The GCC (or Clang) version, configuration, and any patches can all affect how the toolchain finds the rest of itself, and calling it under Bazel can complicate that process. There are notes throughout on common places which will need to be modified, but how they have to be modified requires experimentation with the exact toolchain you want to use. At the end are some debugging tips.

Here is a complete sample project and its testing script.

Writing the BUILD file for the new repository

The compiler is going to be part of a separate repository. This means it needs a BUILD file. That could be included as part of the tarball, but it's usually easiest to keep it separate so changes don't require rebuilding the whole tarball. This BUILD file needs to have filegroups for all the pieces of the compiler tarball which will be used in any other location. Some important ones to get are the tools themselves and all of the include files, shared objects, etc that the toolchain includes. Here's what I have:

```
package(default_visibility = ['//visibility:public'])

filegroup(
  name = 'gcc',
  srcs = [
    'bin/arm-linux-gnueabihf-gcc',
  ],
)

filegroup(
  name = 'ar',
  srcs = [
    'bin/arm-linux-gnueabihf-ar',
  ],
)

filegroup(
  name = 'ld',
  srcs = [
    'bin/arm-linux-gnueabihf-ld',
  ],
)

filegroup(
  name = 'nm',
  srcs = [
    'bin/arm-linux-gnueabihf-nm',
  ],
)

filegroup(
  name = 'objcopy',
  srcs = [
    'bin/arm-linux-gnueabihf-objcopy',
  ],
)

filegroup(
  name = 'objdump',
  srcs = [
    'bin/arm-linux-gnueabihf-objdump',
  ],
)

filegroup(
  name = 'strip',
  srcs = [
    'bin/arm-linux-gnueabihf-strip',
  ],
)

filegroup(
  name = 'as',
  srcs = [
    'bin/arm-linux-gnueabihf-as',
  ],
)

filegroup(
  name = 'compiler_pieces',
  srcs = glob([
    'arm-linux-gnueabihf/**',
    'libexec/**',
    'lib/gcc/arm-linux-gnueabihf/**',
    'include/**',
  ]),
)

filegroup(
  name = 'compiler_components',
  srcs = [
    ':gcc',
    ':ar',
    ':ld',
    ':nm',
    ':objcopy',
    ':objdump',
    ':strip',
    ':as',
  ],
)
```

I saved that file as `compilers/linaro_linux_gcc_4.9.BUILD` under my workspace and then added this fragment to the WORKSPACE file to use it:

```
  new_http_archive(
    name = 'linaroLinuxGcc49Repo',
    build_file = 'compilers/linaro_linux_gcc_4.9.BUILD',
    url = 'https://releases.linaro.org/15.05/components/toolchain/binaries/arm-linux-g
    strip_prefix = 'gcc-linaro-4.9-2015.05-x86_64_arm-linux-gnueabihf',
  )
```

Writing the wrapper scripts

CROSSTOOL refers to the tools using relative paths which may not contain any `..` s, which means wrapper scripts in the same package are necessary. These wrapper scripts are also good places to set LD_LIBRARY_PATH if the compiler needs to refer to shared objects distributed with the binaries. Some toolchains also require setting other environment variables (such as PATH and WIND_BASE). Here's the wrapper script I have for gcc, which is saved as `tools/arm_compiler/linaro_linux_gcc/arm-linux-gnueabihf-gcc` :

```
#!/bin/bash --norc

PATH="external/linaroLinuxGcc49Repo/libexec/gcc/arm-linux-gnueabihf/4.9.3:$PATH" \
  exec \
  external/linaroLinuxGcc49Repo/bin/arm-linux-gnueabihf-gcc \
  "$@"
```

The way I convinced clang to use the correct assembler and linker is by putting them as binaries next to the clang binary itself. This means a separate folder with the clang wrapper script and symlinks (without the target triple on the beginning) to the as and ld wrapper scripts. Here's `tools/arm_compiler/linaro_linux_gcc/clang_bin/clang` :

```
#!/bin/bash --norc

# TODO(Brian): Switch to downloading Clang too.
exec -a "$0" "/usr/bin/clang-3.6" "$@"
```

Here's the wrapper script for nm, saved as `tools/arm_compiler/linaro_linux_gcc/arm-linux-gnueabihf-nm` :

```
#!/bin/bash --norc

exec -a arm-linux-gnueabihf-nm \
  external/linaroLinuxGcc49Repo/bin/arm-linux-gnueabihf-nm \
  "$@"
```

ar, as, cpp, gcov, ld, objcopy, objdump, and strip all have wrapper scripts very similar to the nm one.

Something to note is the use of the -a flag to bash's `exec` builtin. That sets argv[0] of the newly executed image. Some toolchains are picky about whether that is set to the name of the tool with the prefix (like this one), just the name of the tool (ie `exec -a nm ...`), or something including path components.

These wrapper scripts also require a BUILD file. Here's my `tools/arm_compiler/linaro_linux_gcc/BUILD`:

```
package(default_visibility = ['//tools/arm_compiler:__pkg__'])

filegroup(
  name = 'gcc',
  srcs = [
    '@linaroLinuxGcc49Repo//:gcc',
    'arm-linux-gnueabihf-gcc',
  ],
)

filegroup(
  name = 'ar',
  srcs = [
    '@linaroLinuxGcc49Repo//:ar',
    'arm-linux-gnueabihf-ar',
  ],
)

filegroup(
  name = 'ld',
  srcs = [
    '@linaroLinuxGcc49Repo//:ld',
    'arm-linux-gnueabihf-ld',
  ],
)

filegroup(
  name = 'nm',
  srcs = [
    '@linaroLinuxGcc49Repo//:nm',
    'arm-linux-gnueabihf-nm',
  ],
)

filegroup(
  name = 'objcopy',
  srcs = [
    '@linaroLinuxGcc49Repo//:objcopy',
    'arm-linux-gnueabihf-objcopy',
  ],
)

filegroup(
  name = 'objdump',
  srcs = [
    '@linaroLinuxGcc49Repo//:objdump',
    'arm-linux-gnueabihf-objdump',
  ],
)

filegroup(
  name = 'strip',
  srcs = [
    '@linaroLinuxGcc49Repo//:strip',
    'arm-linux-gnueabihf-strip',
  ],
)
```

```
filegroup(
  name = 'as',
  srcs = [
    '@linaroLinuxGcc49Repo//:as',
    'arm-linux-gnueabihf-as',
  ],
)

filegroup(
  name = 'clang',
  srcs = [
    'clang_bin/clang',
  ],
)

filegroup(
  name = 'clang-ld',
  srcs = [
    'clang_bin/ld',
    ':ld',
  ],
)

filegroup(
  name = 'tool-wrappers',
  srcs = [
    ':gcc',
    ':ar',
    ':ld',
    ':nm',
    ':objcopy',
    ':objdump',
    ':strip',
    ':as',
    'clang_bin/as',
    ':clang',
    ':clang-ld',
  ],
)

filegroup(
  name = 'clang-symlinks',
  srcs = glob([
    'clang_more_libs/**',
    'clang_syroot/**',
  ]),
)
```

Writing the CROSSTOOL

Bazel gets most of its information about the C/C++ toolchain from a file called CROSSTOOL.

The `cpu` value for the toolchain is completely arbitrary. It shows up in a few other places, but Bazel doesn't really care what it is. For this Linaro toolchain, the convention is armeabi-v7a.

CROSSTOOL is divided into segments for each CPU+toolchain combination. This is what the section for this toolchain looks like (based on Bazel's default CROSSTOOL for Linux systems):

```
default_toolchain {
  cpu: "armeabi-v7a"
  toolchain_identifier: "clang_linux_armhf"
}

toolchain {
  abi_version: "clang_3.6"
  abi_libc_version: "glibc_2.19"
  builtin_sysroot: ""
  compiler: "clang"
  host_system_name: "armeabi-v7a"
  needsPic: true
  supports_gold_linker: false
  supports_incremental_linker: false
  supports_fission: false
  supports_interface_shared_objects: false
  supports_normalizing_ar: true
  supports_start_end_lib: false
  supports_thin_archives: true
  target_libc: "glibc_2.19"
  target_cpu: "armeabi-v7a"
  target_system_name: "arm_a15"
  toolchain_identifier: "clang_linux_armhf"

  tool_path { name: "ar" path: "linaro_linux_gcc/arm-linux-gnueabihf-ar" }
  tool_path { name: "compat-ld" path: "linaro_linux_gcc/arm-linux-gnueabihf-ld" }
  tool_path { name: "cpp" path: "linaro_linux_gcc/clang_bin/clang" }
  tool_path { name: "dwp" path: "linaro_linux_gcc/arm-linux-gnueabihf-dwp" }
  tool_path { name: "gcc" path: "linaro_linux_gcc/clang_bin/clang" }
  tool_path { name: "gcov" path: "arm-frc-linux-gnueabi/arm-frc-linux-gnueabi-gcov-4
  # C(++) compiles invoke the compiler (as that is the one knowing where
  # to find libraries), but we provide LD so other rules can invoke the linker.
  tool_path { name: "ld" path: "linaro_linux_gcc/arm-linux-gnueabihf-ld" }
  tool_path { name: "nm" path: "linaro_linux_gcc/arm-linux-gnueabihf-nm" }
  tool_path { name: "objcopy" path: "linaro_linux_gcc/arm-linux-gnueabihf-objcopy" }
  objcopy_embed_flag: "-I"
  objcopy_embed_flag: "binary"
  tool_path { name: "objdump" path: "linaro_linux_gcc/arm-linux-gnueabihf-objdump" }
  tool_path { name: "strip" path: "linaro_linux_gcc/arm-linux-gnueabihf-strip" }

  compiler_flag: "-target"
  compiler_flag: "armv7a-arm-linux-gnueabif"
  compiler_flag: "--sysroot=external/linaroLinuxGcc49Repo/arm-linux-gnueabihf/libc"
  compiler_flag: "-mfloat-abi=hard"

  compiler_flag: "-nostdinc"
  compiler_flag: "-isystem"
  compiler_flag: "/usr/lib/clang/3.6/include"
  compiler_flag: "-isystem"
  compiler_flag: "external/linaroLinuxGcc49Repo/lib/gcc/arm-linux-gnueabihf/4.9.3/in
  compiler_flag: "-isystem"
  compiler_flag: "external/linaroLinuxGcc49Repo/arm-linux-gnueabihf/libc/usr/include
  compiler_flag: "-isystem"
  compiler_flag: "external/linaroLinuxGcc49Repo/lib/gcc/arm-linux-gnueabihf/4.9.3/in
  compiler_flag: "-isystem"
  compiler_flag: "external/linaroLinuxGcc49Repo/arm-linux-gnueabihf/libc/usr/include
  cxx_flag: "-isystem"
  cxx_flag: "external/linaroLinuxGcc49Repo/arm-linux-gnueabihf/include/c++/4.9.3/arm
  cxx_flag: "-isystem"
  cxx_flag: "external/linaroLinuxGcc49Repo/arm-linux-gnueabihf/include/c++/4.9.3"
  cxx_flag: "-isystem"
  cxx_flag: "external/linaroLinuxGcc49Repo/include/c++/4.9.3/arm-linux-gnueabihf"
  cxx_flag: "-isystem"
  cxx_flag: "external/linaroLinuxGcc49Repo/include/c++/4.9.3"

  cxx_builtin_include_directory: "%package(@linaroLinuxGcc49Repo)%//include"
  cxx_builtin_include_directory: "%package(@linaroLinuxGcc49Repo)%//arm-linux-gnueab
  cxx_builtin_include_directory: "%package(@linaroLinuxGcc49Repo)%//arm-linux-gnueab
  cxx_builtin_include_directory: "%package(@linaroLinuxGcc49Repo)%//arm-linux-gnueab
  cxx_builtin_include_directory: "%package(@linaroLinuxGcc49Repo)%//include/c++/4.9.
  cxx_builtin_include_directory: "%package(@linaroLinuxGcc49Repo)%//arm-linux-gnueab
  cxx_builtin_include_directory: "%package(@linaroLinuxGcc49Repo)%//arm-linux-gnueab
  cxx_builtin_include_directory: "%package(@linaroLinuxGcc49Repo)%//lib/gcc/arm-linu
  cxx_builtin_include_directory: "%package(@linaroLinuxGcc49Repo)%//lib/gcc/arm-linu
  cxx_builtin_include_directory: "%package(@linaroLinuxGcc49Repo)%//arm-linux-gnueab
  cxx_builtin_include_directory: '/usr/lib/clang/3.6/include'

  linker_flag: "-target"
  linker_flag: "armv7a-arm-linux-gnueabif"
  linker_flag: "--sysroot=external/linaroLinuxGcc49Repo/arm-linux-gnueabihf/libc"
  linker_flag: "-lstdc++"
  linker_flag: "-Ltools/arm_compiler/linaro_linux_gcc/clang_more_libs"
  linker_flag: "-Lexternal/linaroLinuxGcc49Repo/arm-linux-gnueabihf/lib"
  linker_flag: "-Lexternal/linaroLinuxGcc49Repo/arm-linux-gnueabihf/libc/lib"
  linker_flag: "-Lexternal/linaroLinuxGcc49Repo/arm-linux-gnueabihf/libc/usr/lib"
  linker_flag: "-Bexternal/linaroLinuxGcc49Repo/arm-linux-gnueabihf/bin"
```

```
        linker_flag: "-Wl,--dynamic-linker=/lib/ld-linux-armhf.so.3"

        # Anticipated future default.
        # This makes GCC and Clang do what we want when called through symlinks.
        unfiltered_cxx_flag: "-no-canonical-prefixes"
        linker_flag: "-no-canonical-prefixes"

        # Make C++ compilation deterministic. Use linkstamping instead of these
        # compiler symbols.
        unfiltered_cxx_flag: "-Wno-builtin-macro-redefined"
        unfiltered_cxx_flag: "-D__DATE__=\"redacted\""
        unfiltered_cxx_flag: "-D__TIMESTAMP__=\"redacted\""
        unfiltered_cxx_flag: "-D__TIME__=\"redacted\""

        # Security hardening on by default.
        # Conservative choice; -D_FORTIFY_SOURCE=2 may be unsafe in some cases.
        # We need to undef it before redefining it as some distributions now have
        # it enabled by default.
        compiler_flag: "-U_FORTIFY_SOURCE"
        compiler_flag: "-fstack-protector"
        compiler_flag: "-fPIE"
        linker_flag: "-pie"
        linker_flag: "-Wl,-z,relro,-z,now"

        # Enable coloring even if there's no attached terminal. Bazel removes the
        # escape sequences if --nocolor is specified.
        compiler_flag: "-fdiagnostics-color=always"

          # All warnings are enabled. Maybe enable -Werror as well?
        compiler_flag: "-Wall"
        # Enable a few more warnings that aren't part of -Wall.
        compiler_flag: "-Wunused-but-set-parameter"
        # But disable some that are problematic.
        compiler_flag: "-Wno-free-nonheap-object" # has false positives

        # Keep stack frames for debugging, even in opt mode.
        compiler_flag: "-fno-omit-frame-pointer"
```

```
    # Stamp the binary with a unique identifier.
    linker_flag: "-Wl,--build-id=md5"
    linker_flag: "-Wl,--hash-style=gnu"

    compilation_mode_flags {
      mode: DBG
      # Enable debug symbols.
      compiler_flag: "-g"
    }
    compilation_mode_flags {
      mode: OPT

      # No debug symbols.
      # Maybe we should enable https://gcc.gnu.org/wiki/DebugFission for opt or
      # even generally? However, that can't happen here, as it requires special
      # handling in Bazel.
      compiler_flag: "-g0"

      # Conservative choice for -O
      # -O3 can increase binary size and even slow down the resulting binaries.
      # Profile first and / or use FDO if you need better performance than this.
      compiler_flag: "-O2"

      # Disable assertions
      compiler_flag: "-DNDEBUG"

      # Removal of unused code and data at link time (can this increase binary size in
      compiler_flag: "-ffunction-sections"
      compiler_flag: "-fdata-sections"
      linker_flag: "-Wl,--gc-sections"
    }
  }
```

Writing the BUILD file

The BUILD file next to the CROSSTOOL is what ties everything together. The semantics of what files go where is a bit confusing, so it often involves some trial and error. In particular, which compiler tools call which other ones can be surprising. Here's what my tools/arm_compiler/BUILD looks like:

```
# This is the entry point for --crosstool_top.
#
# The cc_toolchain rule used is found by:
#
# 1. Finding the appropriate toolchain in the CROSSTOOL file based on the --cpu
#    and --compiler command line flags (if they exist, otherwise using the
#    "default_target_cpu" / "default_toolchain" fields in the CROSSTOOL file)
# 2. Concatenating the "target_cpu" and "compiler" fields of the toolchain in
#    use and using that as a key in the map in the "toolchains" attribute
cc_toolchain_suite(
  name = 'toolchain',
  toolchains = {
    'armeabi-v7a|clang': ':cc-compiler-armeabi-v7a',
  },
)

filegroup(
  name = 'linaro_linux_all_files',
  srcs = [
    '//tools/arm_compiler/linaro_linux_gcc:clang-symlinks',
    '//tools/arm_compiler/linaro_linux_gcc:tool-wrappers',
    '@linaroLinuxGcc49Repo//:compiler_pieces',
  ],
)

filegroup(
  name = 'linaro_linux_linker_files',
  srcs = [
    '//tools/arm_compiler/linaro_linux_gcc:gcc',
    '//tools/arm_compiler/linaro_linux_gcc:ld',
    '//tools/arm_compiler/linaro_linux_gcc:ar',
    '//tools/arm_compiler/linaro_linux_gcc:clang-ld',
    '//tools/arm_compiler/linaro_linux_gcc:clang',
    '@linaroLinuxGcc49Repo//:compiler_pieces',
  ],
)

filegroup(
  name = 'linaro_linux_compiler_files',
  srcs = [
    '//tools/arm_compiler/linaro_linux_gcc:gcc',
    '//tools/arm_compiler/linaro_linux_gcc:ld',
    '//tools/arm_compiler/linaro_linux_gcc:clang',
    '//tools/arm_compiler/linaro_linux_gcc:as',
  ],
)

cc_toolchain(
  name = 'cc-compiler-armeabi-v7a',
  all_files = ':linaro_linux_all_files',
  compiler_files = ':linaro_linux_compiler_files',
  cpu = 'armeabi-v7a',
  dwp_files = ':empty',
  dynamic_runtime_libs = [':empty'],
  linker_files = ':linaro_linux_linker_files',
  objcopy_files = '//tools/arm_compiler/linaro_linux_gcc:objcopy',
  static_runtime_libs = [':empty'],
  strip_files = '//tools/arm_compiler/linaro_linux_gcc:strip',
  supports_param_files = 1,
)
```

Using the new toolchain

Building with this new toolchain looks like `bazel build --crosstool_top=//tools/arm_compiler:toolchain --cpu=armeabi-v7a`. You can differentiate between multiple toolchains for the same target using --compiler=.

The toolchain for tools that are built and executed during the compilation is set by `--host_crosstool_top`.

TODO(Brian): Actually try this example as-is without any tweaks.

Debugging tips

Figuring out how to get a particular toolchain to work with Bazel can be tricky. Some useful tips are:

- Build with `--verbose_failures` so you can see exactly what command Bazel is running and play with it yourself.
- Build with `--sandbox_debug` to make sure all of the necessary files are getting included in the sandbox.
- Add `compiler_flag: "-v"` to your CROSSTOOL so the compiler will print out more information about the commands it's actually running.