Modular cc toolchains

Please read Bazel Code of Conduct before commenting.

- **Authors:** nathaniel.brough@gmail.com (github: silvergasp)
- **Status: Draft** | In review | Approved | Rejected | In progress | Implemented
- **Reviewers:** tpudlik@google.com
- **Created**: 2023-01-11
- **Updated**: 2023-01-17
- **Discussion thread**: <link>

Note from the Author: I am the maintainer/author of bazelembedded/rules_cc_toolchain, so I may reference work done there, more so that I do with repositories like cfrantz/crt. This is simply a reflection of what I'm familiar with, rather than a value judgement on each of these repositories.

Overview

The current starlark C/C++ toolchain API is sub-optimal and inherits a lot of legacy behaviour from the old CROSSTOOL proto-text configuration files. Creating a new toolchain requires a significant amount of boilerplate and creating a modular framework for defining a custom toolchain is incredibly challenging. The tight coupling in the current toolchain API results in centralised toolchain configurations where small modifications to a toolchain configuration require a complete fork/branch off a centralised toolchain. There have been at least two independent efforts to improve this API by creating a set of abstractions and rules, to allow for a more modular definition of toolchains. Both of these repositories use approximately the same approach to create a modular toolchain.

- bazelembedded/rules_cc_toolchain
- cfrantz/crt

The obvious question is why we need to have extensible and modular toolchains. What are the shortcomings of having a set of centralised toolchains? The answer to this question is not entirely obvious straight off the bat. But here are some of the current challenges;

1. Duplication of code, between a combinatorial explosion of different chip configurations and the corresponding toolchains.
2. Making toolchain easily modifiable, without requiring specialised bazel knowledge.

Duplication of code

Massive duplication of code, for embedded/cross-compilation toolchains using Bazel. This is particularly acute for deeply embedded/bare-metal devices that in general have a near-infinite combination of CPU's, FPU's, SIMD processors, DSP's etc. These combinations can't easily be expressed in a centralised toolchain using Bazel's current constraint_{setting/value} system without a combinatorial explosion of independent toolchains.

The other hitch that comes up here is the need for downstream users to be able to select specific versions and flavours of the underlying tools (i.e. compiler, linker, archiver etc.). Currently it's not possible to have a toolchain spanning across multiple repositories. This is one of the primary reasons why most toolchains maintainers tend towards a single monolithic toolchain rather than one that is modular.

Where are we now?

Currently, most of the toolchain configuration is handled with a set of customized opaque starlark "configuration" rules. A good example of this is shown in the toolchain tutorial, minimised excerpt below;

```
# toolchain/cc_toolchain_config.bzl:
# NEW
load("@bazel_tools//tools/cpp:cc_toolchain_config_lib.bzl", "tool_path")

def _impl(ctx):
    tool_paths = [ # NEW
        tool_path(
            name = "gcc",
            path = "/usr/bin/clang",
        ),
        tool_path(
            name = "ld",
            path = "/usr/bin/ld",
        ),
        tool_path(
            name = "ar",
            path = "/usr/bin/ar",
        ),
        tool_path(
            name = "cpp",
            path = "/bin/false",
        # ...
    ]
    features = [ # NEW
```

```
            feature(
                name = "default_linker_flags",
                enabled = True,
                flag_sets = [
                    flag_set(
                        actions = all_link_actions,
                        flag_groups = ([
                            flag_group(
                                flags = [
                                    "-lstdc++",
                                ],
                            ),
                        ]),
                    ),
                ],
            ),
        ]

    return cc_common.create_cc_toolchain_config_info(
        ctx = ctx,
        # ...
        tool_paths = tool_paths, # NEW
        features = features,
    )
```

This particular method of configuration is conducive towards a large centralised configuration rule. Further, there are some issues that are carried over from the legacy configuration system;

- The tool paths are either absolute paths or they are relative to the instantiation of the configuration rule itself. This is an issue for creating any sort of hermetic toolchain where the downloaded paths cannot create that relative path structure.
- The "tool_paths" key value system makes a set of assumptions that are in line with GCC/clang compilers. Many other compilers will have a different pipeline when building software leading to odd scenarios, where tool paths and flags are not intuitive e.g.

```
tool_path(
  name = "gcc",
  # gcc == clang ???
  path = "/usr/bin/clang",
),
```

- Defining simple compilation/linker flags takes an astonishing amount of boilerplate i.e. 16 lines of code for a single linker flag.

- Each set of features/compiler flags is directly coupled to the toolchain, and they are configured in the custom toolchain configuration rules, there is limited opportunity for re-use. Or we can't express that a re-usable set of flags is for example compatible with only clang, GCC, but not msvc.

Current workarounds

The tool_path problem (tracking issue: #7746)

The "wrapper script"->"injected environment variable" workaround;

This workaround involves creating a directory structure like so;

- "cc_toolchain/BUILD.bazel" (has cc_toolchain_config rule instantiated) e.g. https://github.com/bazelembedded/rules_cc_toolchain/blob/main/cc_toolchain/BUILD.bazel#L92
- "cc_toolchain/wrappers/{gcc,ar,ld...} bash scripts that redirect the compulsory relative toolpath to one that is relative to the output_base directory (bazel info output_base). This can be done by injecting an environment variable into the build. e.g.

```
set -euo pipefail
# Workaround to replace all system includes with user includes
$CPP_TOOL_PATH $@
```

Note: In addition to this approach it's more common to see a hard-coded toolchain wrapper, something like;

```
set -euo pipefail
# Workaround to replace all system includes with user includes
external/gcc_arm_none_compiler/bin/cpp $@
```

The action_config workaround

This workaround was available after the introduction of #10967. While this particular addition is not documented, this change allows the following.

Use the action_config method combined with the separate tool constructor.

From the docstring;

Describes a tool associated with a crosstool action config. Args: path: Location of the tool; Can be absolute path (in case of non hermetic toolchain), or path relative to the cc_toolchain's package. If this parameter is set, tool must not be set. tool: The built-artifact that should be used as this tool. If this is set, path must not be set.

So this looks something like this;

```
action_config (
    config_name = ACTION_NAMES.cpp_link_executable,
    action_name = ACTION_NAMES.cpp_link_executable,
    tools = [
        # NEW label type tool field introduced in #10967
        tool(tool = ctx.executable.my_ld),
    ],
)
```

To get this working in your toolchain you need to go through and create an action_config for each of the ACTION_NAMES. Then you pass your list of action_configs to your toolchain_config.

It's worth noting that while there is some extra boilerplate here there is also extra flexibility in defining which tools get used and when. There are well-defined examples of that in the bazel docs.

It's interesting as well that there are now two entirely separate ways of defining tool targets. The action_config method and the tool_path method.

Relevant links:

- https://github.com/bazelbuild/bazel/issues/7746
- https://stackoverflow.com/questions/73504780/bazel-reference-binaries-from-packages-in-custom-toolchain-definition/73505313#73505313
- https://github.com/bazelbuild/bazel/pull/10967

The feature/flag configuration problem

The challenge here is twofold, reduce boilerplate and create a more-modular and reusable system for defining flags.

Convert a starlark feature into a build rule workaround

This is the approach taken by both;

- bazelembedded/rules_cc_toolchain
- cfrantz/crt

By taking the feature constructor and converting it into a build rule, we can take fully advantage of the configurability semantics that ship with Bazel. This includes select statements, target_compatibility lists etc. This ends up looking something like this;

```
# file: //features:BUILD.bazel
load("@rules_cc_toolchain//cc_toolchain/features/features.bzl",
"cc_feature")
cc_feature(
    name = "garbage_collect_symbols",
    compiler_flags = [
        "-fdata-sections",
        "-ffunction-sections",
    ],
    enabled = True,
    linker_flags = ["-Wl,--gc-sections"],
)
```

But there is no reason why we couldn’t constrain this further to say that this particular feature is only compatible with gcc or clang;

```
# file: //features:BUILD.bazel
load("@rules_cc_toolchain//cc_toolchain/features/features.bzl",
"cc_feature")

constraint_setting(
    name = "compiler_flavour",
)

constraint_value(
    name = "gcc",
    constraint_setting = ":compiler_flavour",
)

constraint_value(
    name = "clang",
    constraint_setting = ":compiler_flavour",
)

cc_feature(
    name = "garbage_collect_symbols",
    compiler_flags = [
        "-fdata-sections",
        "-ffunction-sections",
    ],
    enabled = True,
    linker_flags = ["-Wl,--gc-sections"],
    target_compatible_with = select({
        ":gcc": [],
        ":clang": [],
    }),
)
```

This API can also be further extended to allow for library imports and includes e.g.

```
cc_toolchain_import(
    name = "libc",
    hdrs = glob([inc + "/**/*.h" for inc in INCLUDES] + [inc + "/*.h"
for inc in INCLUDES]),
    additional_libs = [
        "lib/x86_64-linux-gnu/libc.so.6",
        "lib/x86_64-linux-gnu/libc-2.24.so",
        "usr/lib/x86_64-linux-gnu/libc_nonshared.a",
    ],
    includes = INCLUDES,
    runtime_path = "/usr/lib/gcc/x86_64-linux-gnu/6",
    shared_library = "usr/lib/x86_64-linux-gnu/libc.so",
    static_library = "usr/lib/x86_64-linux-gnu/libc.a",
    target_compatible_with = select({
        "@platforms//os:linux": ["@platforms//cpu:x86_64"],
        "//conditions:default": ["@platforms//:incompatible"],
    }),
    visibility = ["@rules_cc_toolchain//config:__pkg__"],
    deps = [
        ":gcc",
        ":math",
        ":mvec",
        ":util",
        "@rules_cc_toolchain_config//:compiler_rt",
    ],
)
```

Proposal

Some of the problems above are partially solved in third party repositories and we can use these as a reference moving forward. However, there are still some open questions;

- How do we effectively manage a configurable interface for hermetic (non-system) tools and compilers?
- How do we make toolchain definitions more modular and composable?

The general proposal to create a modular toolchain is to move the toolchain configuration from a per toolchain custom bazel plugin out into BUILD definitions. While this approach has been explored in some detail in the workarounds section, the effort is far from complete.

The important components of the current toolchain API that are proposed to be converted from custom configuration plugins to individual rules include;

- Features

- Action configs

Features

As it currently stands a "feature" definition in the Bazel toolchain, consists of a constructor "feature" that returns a provider "FeatureInfo". The proposal here is to have a simple Bazel rule that takes the feature constructor and uses it to create a simplified Bazel rule. E.g.

```
# file: //my_custom_toolchain:BUILD.bazel

load("@rules_cc//cc/defs.bzl," "cc_feature")
load("@rules_cc//cc/actions.bzl," "ACTION_NAMES")
cc_feature(
  name = "pic",
  enabled = 1,
  action_flags = {
    ACTION_NAMES.assemble: ["-fpic"],
    ACTION_NAMES.preprocess_assemble: ["-fpic"],
    ACTION_NAMES.linkstamp_compile: ["-fpic"],
    ACTION_NAMES.c_compile: ["-fpic"],
    ACTION_NAMES.cpp_compile: ["-fpic"],
  },
)
```

Now you might notice here that we have a lot of duplication here i.e. we have to attach the "-fpic" flag to each of our actions. This is largely due to the limitations on the types that are available in the attr module for Bazel rules. Thankfully we can create a wrapper similar in concept to bazel-skylib's "selects.with_or" macro, in this case, the above could be reduced down to;

```
# file: //my_custom_toolchain:BUILD.bazel

load("@rules_cc//cc/defs.bzl," "cc_feature")
load("@rules_cc//cc/actions.bzl," "ACTION_NAMES", "action_mux")
cc_feature(
  name = "pic",
  enabled = 1,
  action_flags = action_mux({
    (ACTION_NAMES.assemble,
     ACTION_NAMES.preprocess_assemble,
     ACTION_NAMES.linkstamp_compile,
     ACTION_NAMES.c_compile,
     ACTION_NAMES.cpp_compile) : ["-fpic"],
  }),
)
```

Where the action_mux macro simply expands out each action in the tuple to be a complete dictionary like the one above.

Now there is a hitch, we need to be able to handle variable expansion. Variable expansion is required for flags where we don't necessarily have complete information about the flag when resolving the toolchain configuration. Examples of this include;

- Compiler outputs (we can't know the output path for every target ahead of time)
- Include paths (we can't hard code the include paths for every target ahead of time)

So to express variable expansion in our "cc_feature" rule we need to make a compromise to ensure compatibility with our rules. The approach we will take is to restrict each feature to allow for only one variable expansion. This is sufficient for most features, however, it does require workarounds in some cases. Here is an example of an expansion with a single variable;

```
# file: //my_custom_toolchain:BUILD.bazel

load("@rules_cc//cc/defs.bzl," "cc_feature")
load("@rules_cc//cc/actions.bzl," "ACTION_NAMES", "action_mux")
cc_feature(
  name = "preprocessor_defines",
  enabled = 1,
  iterate_over = "preprocessor_defines",
  action_flags = action_mux({
    (ACTION_NAMES.assemble,
     ACTION_NAMES.preprocess_assemble,
     ACTION_NAMES.linkstamp_compile,
     ACTION_NAMES.c_compile,
     ACTION_NAMES.cpp_compile) : ["-D%{preprocessor_defines}"],
  }),
)
```

An example of a feature that would often require multiple variable expansions is the "include_paths" feature. This is because we may have different types of includes that we want to manage independently, and to mirror this there are 4 separate include variables that are available for expansion;

- quote_include_paths
- include_paths
- system_include_paths
- framework_include_paths

One of the disadvantages of this approach is that we won't be able to express multiple expansions under a single feature. Instead, we'll introduce the concept of dependent features, which map onto the "implies" attribute of the current features API.

```
# file: //my_custom_toolchain:BUILD.bazel
```

```
load("@rules_cc//cc/defs.bzl," "cc_feature")
load("@rules_cc//cc/actions.bzl," "ACTION_NAMES", "action_mux")
INCLUDE_ACTIONS = (ACTION_NAMES.assemble,
     ACTION_NAMES.preprocess_assemble,
     ACTION_NAMES.linkstamp_compile,
     ACTION_NAMES.c_compile,
     ACTION_NAMES.cpp_compile)

cc_feature(
  name = "include_paths",
  enabled = 1,
  # NEW deps attribute
  deps = [
    ":quote_includes",
    ":include_paths",
    # ...
  ],
)

cc_feature(
  name = "quote_include_paths",
  iterate_over = "quote_include_paths",
  action_flags = action_mux({
    INCLUDE_ACTIONS: ["-iquote%{quote_include_paths}"],
  }),
)

cc_feature(
  name = "include_paths",
  iterate_over = "include_paths",
  action_flags = action_mux({
    INCLUDE_ACTIONS: ["-I%{include_paths}"],
  }),
)
# ...
```

A further consideration is how we keep track of features that are incompatible with each other. For example, when it comes to compilation mode features “opt”, “dbg”, “fastbuild”, we want to ensure that only one of these features can be applied at any given point. This is currently achieved using the “provides” attribute of the feature constructor. To do this in the new api we’ll introduce a new rule “cc_feature_setting”, which will have similar semantics to Bazel’s built in “constraint_setting”. This will look something like this;

```
# file: //my_custom_toolchain:BUILD.bazel

```

```
load("@rules_cc//cc/defs.bzl," "cc_feature", "cc_feature_setting")
load("@rules_cc//cc/actions.bzl," "ACTION_NAMES", "action_mux")
cc_feature_setting(
  name = "compilation_mode",
)

cc_feature(
  name = "opt",
  feature_setting = ":compilation_mode",
  # ...
)

cc_feature(
  name = "dbg",
  feature_setting = ":compilation_mode",
  # ...
)

cc_feature(
  name = "fastbuild",
  feature_setting = ":compilation_mode",
  # ...
)
```

Open questions

Up until this point there have been no incompatible changes introduced and this is simply proposed as an abstraction layer over the current toolchains API. However, there are some additional areas that we should explore. This includes;

- How do we map the target name to the feature name (in the toolchain). I.e. in the example above do we call the feature "dbg" or "@my_repository//my_toolchain:dbg". What implications would a full label name have for compatibility and consistency in the build files. Would we have to create a set of "common" alias features, so that you can still use the common "features" definitions with a common interface?
- Do we want to create a more consistent API around action names e.g. would we want each action to have its own target i.e. "@rules_cc//actions:c_compile", rather than the constant defined as "ACTION_NAMES.cc_compile". E.g.

```
# file: //my_custom_toolchain:BUILD.bazel

load("@rules_cc//cc/defs.bzl," "cc_feature")
cc_feature(
  name = "pic",
  enabled = 1,
  action_flags = action_mux({
    ("@rules_cc//actions:assemble",
     "@rules_cc//actions:preprocess_assemble",
     # ...
     "@rules_cc//actions:cpp_compile") : ["-fpic"],
  }),
)
```

Action configs

Action configs serve the purpose of coordinating the usage of specific tools with the flags defined in the corresponding features. The proposal here is to take an identical approach to the feature rule. E.g.

```
# file: //my_custom_toolchain:BUILD.bazel

load("@rules_cc//cc/defs.bzl," "cc_action_config", "ACTION_NAMES")

config_setting(
    name = "fastbuild",
    values = {"compilation_mode": "fastbuild"},
)

config_setting(
    name = "opt",
    values = {"compilation_mode": "opt"},
)

config_setting(
    name = "dbg",
    values = {"compilation_mode": "dbg"},
)

alias(
  name = "linker",
  actual = select({
    ":fastbuild": ":fastbuild_linker",
    ":opt": ":opt_linker",
    ":dbg": ":dbg_linker",
  }),
)

cc_action_config(
  name = "opt_linker",
  action_name = ACTION_NAMES.cpp_link_executable,
  # Specify the tool as either a label or an absolute path
  tool = "@com_clang//:bin/ld.lld",
  # tool_path = "/usr/bin/clang",
  # action_flags = {...}
)

cc_action_config(
  name = "fastbuild_linker",
  action_name = ACTION_NAMES.cpp_link_executable,
  tool = "@com_mold//:bin/mold",
  # action_flags = {...}
)

alias(
  name = "dbg_linker",
  actual = ":fastbuild_linker",
)
```

Open questions

At the moment it is not entirely clear to me why we couldn't merge the action_config and the feature API's into one. This would be doable by keeping the tool_path and, tool attributes optional. Would this make sense?

A complete toolchain using the new API

Using the proposed API we'd be able to significantly lower the barrier for entry for creating a new toolchain or slightly modifying one in a modular manner. E.g.

```
cc_action_config(
  name = "linker_action",
  action_name = ACTION_NAMES.cpp_link_executable,
  tool = "@com_llvm//:bin/ld.lld",
)

# ... An action config per action to specify the tool to use ...

cc_feature(
  name = "default_linker_flags",
  enabled = 1,
  action_flags = action_mux({
    (ACTION_NAMES.cpp_link_executable,
      ACTION_NAMES.cpp_link_dynamic_library,
      ACTION_NAMES.cpp_link_nodeps_dynamic_library): ["-lstdc++"],
  })
)

# ... Other features ...

cc_toolchain_config(
  name = "my_toolchain_config",
  action_configs = [
    ":linker_action",
    # ... Other actions configs ...
  ],
  features = [
    ":default_linker_flags",
    # ... Other Features ...
  ],
)

cc_toolchain(
   name = "my_toolchain",
   config = ":my_toolchain_config",
   # ... Other file attributes ...
)
```

It's worth noting here as well is that the cc_toolchain_config rule, which is usually provided by the toolchain maintainer, probably doesn't need to exist anymore and could probably be merged with the cc_toolchain target or bundled up together using a macro.

Ergonomics and re-usability

To fully take advantage of the new "toolchain feature in the BUILD file" we'd Ideally be able to take advantage of the full set of configuration tools that are available in the Bazel build environment i.e.;

- selects,
- target_compatibility,
- config_settings etc.

To give a more concrete example it would be nice to be able to constrain a feature to a particular set of compilers. E.g. a feature that supports only gcc and clang but doesn't support any other compiler.

```
cc_feature(
  name = "default_linker_flags",
  enabled = 1,
  action_flags = action_mux({
    (ACTION_NAMES.cpp_link_executable,
      ACTION_NAMES.cpp_link_dynamic_library,
      ACTION_NAMES.cpp_link_nodeps_dynamic_library): ["-lstdc++"],
  }),
  target_compatible_with = select({
    "@rules_cc//cc/compiler_flavour:gcc": [],
    "@rules_cc//cc/compiler_flavour:clang": [],
    "//conditions:default": ["@platforms//:incompatible"],
  })
)
```

Exactly how to implement this step needs to be explored, however, it should be doable to achieve this using transitions. i.e. the toolchain specifies a transition setting for the compiler flavour.

Compatibility

Initially there will be no incompatibilities with the new approach as we will just be creating an abstraction layer over the current API. However, in the process of doing so, we may have the opportunity to deprecate some problematic API's. For example, if we are able to use the action_config approach to track tool location then we should be able to deprecate the redundant and problematic tool_path API. This proposal will maintain backward compatibility, but make recommendations on breaking changes.

Document History

Date	**Description**
2023-01-11	First proposal.
2023-01-14	Removed section on building system libraries from source. As that effort will be tracked by a separate proposal.
2023-01-15	• Add details on compatibility. • Adds a more complete description of the proposed API
2023-01-17	Adds a small section on compiler flavours, and the ability to configure compatibility and selectively set flags on a per feature basis.