[RFC] Modifiers

This document proposes a new configuration feature in Buck called modifiers. The goal is to provide a unified way to specify build settings on a project level, on individual targets, and on the command line and address problems with the current methods of customizations in Buck.

Context

Why do we need a new configuration setup?

A target often needs to be built in multiple build settings. For example, a single target may be customized with different OSes (ex. linux, mac, windows), architectures (ex. x86, arm), and sanitizers (ex. asan, tsan, ubsan). Buck has 2 main ways of supporting customizations today:

1. Buckconfigs specified through `--config` or `-c` flags. They are global flags and are often aggregated in modefiles (`@<modefile>` on the command line).
2. Platforms specified through `default_target_platform` attribute or `--target-platforms` flag), which become a target's target "configuration". `--target-platforms` flags are also commonly specified via modefiles.

These methods suffer from the following problems.

1. *High discovery cost and cognitive load.* Many targets don't build out of the box and require a dedicated modefile. It's onerous for users to know the right modefiles to use for a target, and they often don't realize when they are using the wrong modefiles.
2. *Too many modefiles*. A monorepo can end up with a huge number of project-specific modefiles when each customized project adds its own set of modefiles. Internally, the number of modefiles in our monorepo is on the order of **1000s**.
3. *Slow incremental builds*. Changing buckconfigs invalidates Buck's global state and causes Buck to always rerun load and analysis on incremental builds. This adds non-trivial Buck overhead on every incremental build.
4. *Lack of multi-configuration support*. Buck will execute commands with different buckconfig states sequentially. This prevents Buck from building in multiple modes in parallel. Target platforms support multi-configuration builds.
5. *Platform generation is exponential in the number of build settings*. Suppose a repo supports 3 OSes, 2 CPU architectures, and 3 compilers. Platforms require generating all 18 permutations of these settings as targets, which is not scalable.
6. *Platform does not compose well*. Suppose I want to apply ASAN. It's not possible to specify ASAN on top of an existing platform on the command line. Instead, a new platform target must be created based on the existing platform and ASN.
7. *Poor tooling integration*. Similar to users, it's onerous for tooling to keep track of what modes are needed to build a target with. Additionally, buckconfigs are bad for performance for tools like language servers because it's impossible to request builds of

two targets that require different modes to build in parallel.

8. *Breaks repo-wide queries*. Buckconfigs mean that there is not one unified unconfigured target graph but many variants of it based on different modefiles, and different parts of repo may only be queried with certain modefiles. This prevents simple queries like “what targets in the repo depend on this third-party library” from working in practice.

Modifier API Goals

The Modifier API introduces a unified way to specify build settings on a target and on the command line. Like target platforms, it constructs Buck configurations so it supports multi-configuration builds. Our goals with modifiers are as follows.

1. `buck build` on any target should work without extra flags like `@mode` or `--config`.
2. `buck build` on any target should work with a small (~10s) set of unified modes like debug, Linux, and asan.
3. Delete thousands of modefiles in repo.
4. Eliminate Buck overhead when changing build settings.
5. Simplify tooling support for tools like language servers.
6. Make repo-wide queries functional for users and tooling and improve CI effectiveness.

Configuration Background

Feel free to skip this if you already understand Buck configurations.

A configuration is a collection of `constraint_value` targets. Each individual constraint value is keyed by a `constraint_setting` (commonly referred to as just constraint), so there can only be one constraint value of a constraint in a configuration.

For example, the following BUCK file defines `prelude//constraints/os:_` as a constraint setting with constraint values `prelude//constraints/os:linux`, `prelude//constraints:macos`, and `prelude//constraints:windows`.

```python
# prelude//constraints/os/BUCK

constraint_setting(name = "os")

constraint_value(
  name = "linux",
  constraint_setting = ":os",
)

constraint_value(
  name = "macos",
```

```
  constraint_setting = ":os",
)

constraint_value(
  name = "windows",
  constraint_setting = ":os",
)
```

A configuration
may contain either `prelude//constraints/os:linux`,
`prelude//constraints/os:macos`, or `prelude//constraints/os:windows` to
indicate which OS a target is built for.

A constraint or a set of constraints can be matched based on the `select` operator to
customize a target's behavior. For example, the following adds a linux only dep
to a target.

```
Python
deps = select({
  "prelude//constraints/os:linux": [":linux_only_dep"],
  "DEFAULT": [],
})
```

Before building a target specified from the command line (known as a top-level target), Buck needs to know its configuration in order to resolve selects. Modifiers are a new way to resolve a target's configuration for every top-level target.

Note: throughout this doc we will use targets like `prelude//constraints/os:linux` to indicate constraint values without defining them in examples. To use any constraint in practice, you have to define the `constraint_setting` and `constraint_value` targets first.

API

Under modifiers, every top-level target will start with an empty configuration, and Buck will apply a list of "modifiers" in a specific order to obtain a configuration. A modifier is a modification of a specific constraint in the configuration. There are two types of modifiers, *conditional* and *unconditional* modifiers.

An unconditional modifier is just a constraint value. Applying an unconditional modifier will insert the associated constraint value into the configuration for the respective constraint, replacing any existing constraint value for that setting. For example, specifying `prelude//constraints/os:windows` as a modifier will insert `prelude//constraints/os:windows` into the configuration and override existing constraint value for the `prelude//constraints/os:_` constraint setting.

A conditional modifier is a modifier that only applies when a certain condition is satisfied. This lets one express powerful composition based on other criteria. `modifiers.conditional()` is a conditional modifier that changes the constraint value inserted based on the existing configuration. For example, a modifier like

```
Python
modifiers.conditional({
  "prelude//constraints/os:windows": "prelude//constraints/compiler:msvc",
  "DEFAULT": "prelude//constraints/compiler:clang",
})
```

will insert MSVC constraint into the configuration if the OS is windows or clang constraint otherwise. A `modifiers.conditional` behaves similarly to Buck's `select` but can only be used in a modifier context.

A `modifiers.conditional` can only be used to modify a single constraint, so the following example is not valid.

```
Python
# This fails because a modifier cannot modify both compiler and OS.
modifiers.match({
  "prelude//constraints/os:windows": "prelude//constraints/compiler:msvc",
  "DEFAULT": "prelude//constraints/os:linux",
})
```

A modifier can be specified in a PACKAGE file, on a target, or on the command line. This provides the flexibility needed to customize targets on a project, target, or cli level.

Per-Tree Modifier

In a PACKAGE or BUCK_TREE file, modifiers can be specified using the `set_cfg_modifiers` function and would apply to all targets covered under that PACKAGE or BUCK_TREE file. For example, modifiers specified in `repo//PACKAGE` would apply to any

target under `repo//...`. Modifiers specified in `repo/foo/PACKAGE` would apply to any target under `repo//foo/...` (for resolution order, see "Modifier Resolution" section).

The `set_cfg_modifiers` function takes as input a list of modifiers. The following is an example that sets modifiers for OS and compiler settings for all targets in the repo.

```
Python
# repo/PACKAGE

set_cfg_modifiers(cfg_modifiers = [
  "prelude//constraints/os:linux",
  modifiers.match({
    "DEFAULT": "prelude//constraints/compiler:clang",
    "prelude//constraints/os:windows": "prelude//constraints/compiler:msvc",
  }),
])
```

Per-Target Modifier

On a target, modifiers can be specified on the `modifiers` attribute. For example, the following specifies modifiers for `repo//foo:bar`.

```
Python
# repo/foo/BUCK

python_binary(
  name = "bar",
  # ...
  modifiers = [
    "prelude//constraints/os:windows",
    "prelude//constraints/compiler:clang",
  ],
)
```

Note that for legacy reasons, we also support modifiers defined on `metadata` attribute via “buck.cfg_modifiers” key.

```
Python
# repo/foo/BUCK

python_binary(
  name = "bar",
```

```
    # ...
    metadata = {"buck.cfg_modifiers": [
      "cfg//os:windows",
      "prelude//constraints/compiler:clang",
    ]},
)
```

We are in the process of migrating these use cases to the `modifiers` attribute. In the meantime, if a target has both `modifiers` and metadata key “buck.cfg_modifiers” defined, Buck will throw an error at configuration time.

Input Modifier

On the command line, modifiers are specified as `buck2 build <target>?<modifiers separated by commas>`. For example, `buck2 build repo//foo:bar?prelude//constraints/sanitizer:asan` applies asan modifier on the command line. `buck2 build repo//foo:bar?prelude//constraints/os:linux,prelude//constraints/sanitizer:asan` will apply linux and asan modifiers.

To make constraints easier to type, alias strings can be specified for modifier targets and used on the command line. `buck2 build repo//foo:bar?asan` is valid provided the following aliases are specified.

```
Unset
[alias]
  asan = prelude//constraints/sanitizer:asan
```

Modifiers can be specified for any target pattern, so `buck2 build repo//foo/...?asan` and `buck2 build repo//foo:?asan` are both valid.

When specifying a subtarget and modifier with `?`, subtarget should go before the modifier, ex. `buck2 build repo//foo:bar[comp-db]?asan`.

To specify modifiers to a list of target patterns on the command line, you can use the `--modifier` or `-m` flag. For example, `buck2 build repo//foo:bar repo//foo:baz -m release` is equivalent to `buck2 build repo//foo:bar?release //foo:baz?release`.

`--modifier` flag can be specified multiple times to add multiple modifier, so

`buck2 build --modifier=linux --modifier=release repo//foo:bar` is equivalent to `buck2 build repo//foo:bar?linux,release`.

It is prohibited to specify both `--modifier` flag and `?` in a target pattern. This restriction can be lifted in the future if there is a need.

When two modifiers of the same constraint setting are specified, then the later one overrides the earlier one. For example, `buck2 build repo//foo:bar?dev,release` is equivalent to `buck2 build repo//foo:bar?release`.

On command line, a `config_setting` target can be specified as a collection of modifiers after `--modifier` or `?`. This will be equivalent to specifying each constraint value inside the `config_setting` as a separate modifier.

NOTE: only `--modifier` and `-m` flags are currently implemented. `?` is not implemented.

Modifier Resolution

Modifiers are applied in order of constraint setting, and for each constraint setting, modifiers for that setting are resolved in order of PACKAGE, target, and command line, with modifiers from parent PACKAGE applied before child PACKAGE. The end of this section will describe how Buck determines the order of constraint setting to resolve.

Suppose modifiers for repo//foo:bar are specified as follows.

Python

```
# repo/PACKAGE

set_cfg_modifiers(cfg_modifiers = [
  "cfg//:linux",
  modifiers.match({
    "DEFAULT": "cfg//compiler:clang",
    "cfg//os:windows": "cfg//compiler:msvc",
  }),
])

# repo/foo/PACKAGE

set_cfg_modifiers(cfg_modifiers = ["cfg//os:macos"])

# repo/foo/BUCK

python_binary(
```

```
    name = "bar",
    # ...
    metadata = {"buck.cfg_modifiers": ["cfg//os:windows"]},
)
```

At the beginning, the configuration will be empty. When resolving modifiers, Buck will first resolve all modifiers for `cfg//os:_` before resolving all modifiers for `cfg//compiler:_`.

For OS, the linux modifier from `repo/PACKAGE` will apply first, followed by macos modifier from `repo/foo/PACKAGE` and windows modifier from `repo//foo:bar`'s target modifiers, so `repo//foo:bar` will end up with `prelude//constraints/os:windows` in its configuration. Next, to resolve the compiler modifier, the `modifiers.conditional` from `repo/PACKAGE` will resolve to `prelude//constraints/compiler:msvc` since the existing configuration is windows and apply that as the modifier. The target configuration for `repo//foo:bar` ends up with windows and msvc.

However, suppose the user requests `repo//foo:bar?linux` on the command line. When resolving the OS modifier, the linux modifier from cli will override any existing OS constraint and insert linux into the configuration. Then, when resolving the compiler modifier, the `modifiers.conditional` will resolve to `prelude//constraints/compiler:clang`, giving clang and linux as the final configuration.

Because command line modifiers will apply at the end, they are also known as required modifiers. Any modifier specified on the command line will always override any modifier for the same constraint setting specified in the repo.
The ordering of constraint settings to resolve modifiers is determined based on the dependency order of constraints specified in the keys of the `modifiers.conditional` specified. Because some modifiers match on other constraints, modifiers for those constraints must be resolved first. In the previous example, because the compiler modifier matches on OS constraints, Buck will resolve all OS modifiers before resolving compiler modifiers. `modifiers.conditional` that ends up with a cycle of matched constraints (ex. compiler modifier matches on sanitizer but sanitizer modifier also matches on compiler) will be an error.

Host Conditional Modifiers

Modifiers have 2 types of conditional modifiers that allow for powerful compositions. Each operator is a function that accepts a dictionary where the keys are the conditionals and values are modifiers.

1. Conditional modifier. Introduced in the previous sections, this is capable of inserting constraints based on constraints in the existing configuration.

2. Host conditional modifier. This selects based on the host configuration, whereas `modifiers.match` selects based on the target configuration. This host configuration is constructed when resolving modifiers. `modifiers.match_host` is important to making `buck build` work anywhere on any platform. For example, when the OS to configure is not specified, it's best to assume that the user wants to target the same OS as the host machine.

NOTE: host conditional modifiers are currently not implemented.

An example is roughly as follows.

```
Python
# root/PACKAGE

# We want OS to target the host machine by default.
# Ex. build linux on linux machine, build windows on windows machine,
# and build mac on mac machine.
# However, if the rule is apple or android specific, then we should
# always be building for apple/android as OS, no matter the host
# configuration.

set_cfg_modifiers(cfg_modifiers = [
  modifiers.match_rule({
    "apple_.*": "cfg//os:iphone",
    "android_.*": "cfg//os:android",
    "DEFAULT": host_select({
      "cfg//os:linux": "cfg//os:linux",
      "cfg//os:macos": "cfg//os:macos",
      "cfg//os:windows": "cfg//os:windows",
    }),
  }),
])
```

On select resolution, Buck's select currently requires unambiguous keys in the dictionary and resolves to the key with the most refined match. The select operators used in modifiers will diverge from this and implement a "first-match" behavior, where select resolves to the first condition that evaluates to true in the dictionary.

Legacy Target platform

Target platform (`--target-platforms` flag or `default_target_platform` attribute) will be a deprecated way of specifying configuration and will be killed once all use cases migrate to modifiers. To maintain backwards compatibility with target platforms during the migration process, modifier resolution will take into account the target platform specified. This allows for

an easy migration where modifiers can be introduced one at a time without reaching feature parity of target platforms.

If a target's modifiers resolve to an empty configuration, then Buck will reuse the target platform as the configuration. If modifiers resolve to a non-empty configuration, then Buck looks for any constraint in the target platform not covered by a constraint setting from the modifier configuration and adds those to the configuration. For example, suppose in the previous example, the target platform for `repo//foo:bar` includes `cfg//sanitizer:asan`, then this constraint will be inserted into the configuration since no modifier covered the sanitizer constraint setting.

Debugging modifiers

NOTE: Below is unimplemented.

Because many layers of modifiers can be applied before obtaining a final configuration, it is important that modifier resolution is easy to debug and understand. Here are some ways that modifier resolution can be interpreted.

1. *`buck2 audit modifiers` command*. There will be a `buck2 audit modifiers` command to show all PACKAGE, target, and required modifiers for a target. It can also show configuration changes from the modifier resolution process if requested by the user.
2. *Starlark print or debugger support*. Modifier resolution process will be implemented in Starlark in prelude. This means that any user can use any of the existing way to debug starlark (ex. print statements, Starlark debugger in VSCode) to debug the resolution process.

How configuration modifiers differ from transitions

Modifiers are largely inspired by configuration transitions. Currently, the difference between the two is that a transition can change the configuration of any target in the graph, but a modifier can only change the configuration of a top-level target. In other words, if you have target A that depends on target B and you request a build of A, then A's target configuration would be resolved via modifiers and propagated down to B, but dep B would not do its own modifier resolution.

We are looking at adding support for modifier transition, which can enable transition via modifiers, but that is out of the scope of this RFC.

For now, we recommend using transitions for any configuration that needs to be applied on dependencies and modifiers otherwise. Some examples include:

1. *Python version* should be modeled as a transition and not a modifier. Suppose we have `python_binary` A nested as a resource of another `python_binary` B. A should not inherit the python version from B, so a transition is needed to change A's python version

when depended on by B.

2. *Library targets* should use modifiers and not transitions. A C++ library target should always inherit the configuration of its parent C++ binary when it is used as a dep, but a top-level C++ library target can still have its configuration changed via modifiers when requested from the command line.

Link to google doc: [RFC] Modifiers