How GN handles cross-compiling

As a GN user

GN has robust support for doing cross compiles and building things for multiple architectures in a single build (e.g., to build some things to run locally and some things to run on an embedded device). In fact, there is no limit on the number of different architectures you can build at once; the Chromium build uses at least four in some configurations.

To start, GN has the concepts of a *host* and a *target*. The host is the platform that the build is run on, and the target is the platform where the code will actually run (This is different from autotools' terminology, but uses the more common terminology for cross compiling**).**

(Confusingly, GN also refers to each build artifact – an executable, library, etc. – as a target. On this page, we will use "target" only to refer to the system you want to run your code on, and use "rule" or some other synonym to refer to a specific build artifact).

When GN starts up, the `host_os` and `host_cpu` variables are set automatically to match the operating system (they can be overridden in args files, which can be useful in weird corner cases). The user can specify that they want to do a cross-compile by setting either or both of `target_os` and `target_cpu`; if they are not set, the build config files will usually set them to the host's values, though the Chromium build will set target_cpu to "arm" if target_os is set to "android").

So, for example, running on an x64 Linux machine:

```
gn gen out/Default
```

is equivalent to:

```
gn gen out/Default --args='target_os="linux" target_cpu="x64"'
```

To do an 32-bit ARM Android cross-compile, do:

```
gn gen out/Default --args='target_os="android"'
```

(We don't have to specify target_cpu because of the conditionals mentioned above).

And, to do a 64-bit MIPS ChromeOS cross-compile:

```
gn gen out/Default --args='target_os="chromeos" target_cpu="mips64el"'
```

As a BUILD.gn author

If you are editing build files outside of the //build directory (i.e., not directly working on toolchains, compiler configs, etc.), generally you only need to worry about a few things:

The `current_toolchain`, `current_cpu`, and `current_os` variables reflect the settings that are **currently** in effect in a given rule. The `is_linux`, `is_win` etc. variables are updated to reflect the current settings, and changes to `cflags`, `ldflags` and so forth also only apply to the current toolchain and the current thing being built.

You can also refer to the `target_cpu` and `target_os` variables. This is useful if you need to do something different on the host depending on which target_arch is requested; the values are constant across all toolchains. You can do similar things for the `host_cpu` and `host_os` variables, but should generally never need to.

By default, dependencies listed in the `deps` variable of a rule use the same (currently active) toolchain. You may specify a different toolchain using the `foo(bar)` label notation as described in GNLanguage#Labels.

As a //build/config or //build/toolchain author

As described in GNLanguage#Overall-build-flow, the `default_toolchain` is declared in the `//build/config/BUILDCONFIG.gn` file. Usually the `default_toolchain` should be the toolchain for the `target_os` and `target_cpu`. The `current_toolchain` reflects the toolchain that is currently in effect for a rule.

Be sure you understand the differences between `host_cpu`, `target_cpu`, `current_cpu`, and `toolchain_cpu` (and the os equivalents). The first two are set as described above. You are responsible for making sure that `current_cpu` is set appropriately in your toolchain definitions; if you are using the stock templates like `gcc_toolchain` and `msvc_toolchain`, that means you are responsible for making sure that `toolchain_cpu` and `toolchain_os` are set as appropriate in the template invocations.