Measuring Configuration Overhead

Please read Bazel Code of Conduct before commenting.

- **Authors**: Greg Estren (gregce@bazel.build)

- **Status**: Approved

- **Reviewers**: Stephen Twigg (twigg+bazel@google.com)

- **Created**: 2020-02-28
- **Updated**: first draft
- **Tracking bug:** https://github.com/bazelbuild/bazel/issues/10613

Summary

If you use transitions, change flags between builds, or use remote caching, you may experience slower builds than expected.

This document proposes a tool for measuring this impact and understanding root causes - specifically those related to configuration.

Background

Configuration is the mechanism by which Bazel builds targets with different settings.

Prominent examples are building for multiple platforms and building with specific SDKs or product features. These can be set at the command line or with transitions.

The technical difference between configuration and attributes is attributes only change the rule they're attached to, while configuration changes the rule's whole sub-graph.

This has crucial performance consequences. Without optimizations like *trimming*, multi-configured builds can easily create large build graphs with poor caching. This slows down builds and raises build machine memory needs.

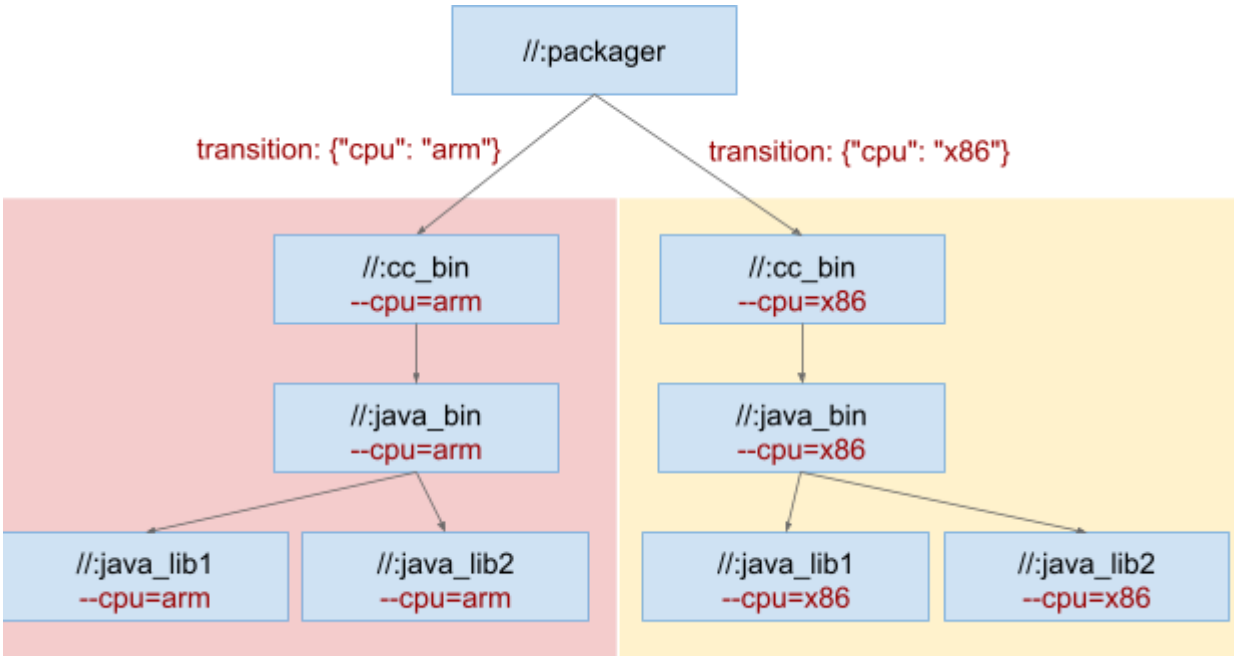
Work is ongoing to minimize these effects. But it's also important to understand *why* they happen, beyond unactionable observations like "*I added a transition and my build time doubled*". It should be possible to analyze configuration like any other resource-affecting behavior and derive actionable conclusions to keep builds fast and lean.

This document proposes a mental model and measurement tool to support this.

Example

Imagine a rule that packages two instances of the same C++ binary for different CPUs. The binary also has extensive Java dependencies due to a source generation tool.

Bazel's configured target graph will look something like this:



The key insight is that every rule except `//:packager` is cloned for each CPU even though the Java rules don't depend on the CPU. This makes the build graph nearly twice as large as it needs to be.

Bazel also *universally* adds the CPU to all output file paths, for example `bazel-out/x86-fastbuild/bin/java_bin.jar`. This means even Java compile actions can't share results from an executor cache.

One way to optimize this build is to remove or pre-compile the Java binary. Another is to "trim" the CPU from the Java rule's configurations so they can be merged together.

We'd like Bazel users to be able to extract this level of insight from real-life builds.

Proposal

This document proposes a command-line tool that reads cquery's --show_config_fragments output to show what "pieces"'of configuration rules require and which dependencies create those requirements.

This establishes a mental model for understanding how build graphs consume configuration. This enables programmatic queries that can answer pointed questions about a build's efficiency and bottlenecks.

This tool already exists experimentally, as a Python program called `ctexplain`. This proposal both shows what `ctexplain` does and suggests a path for productionizing it.

cquery --show_config_fragments

Configuration consists of domain-specific sub-components. For example, a `select()` on `--cpu` must know the target CPU but not C++ link options. Language-specific settings are generally grouped into common collections known as *fragments* (example).

Rule definitions must declare which fragments they use. This lets Bazel "know" that C++ rules need C++-related settings while Java rules don't. This is crucial information for understanding the conclusions in the "Java binary tool" example.

`cquery's --show_config_fragments` option surfaces this information to the user. It accepts two values:

- `direct:` annotate each rule with its declared configuration requirements
- `transitive:` annotate each rule with *its transitive dependencies'* declared requirements

`transitive` mode is important because even if a rule doesn't directly need a piece of configuration its output still depends on what its transitive dependencies do.

Example:

```
$ bazel cquery --show_config_fragments=transitive 'deps(//:cc_bin)'
//:cc_bin (f59f66e649bce) [CoreOptions, CppConfiguration, JavaConfiguration,
PlatformConfiguration, ShellConfiguration]
//:java_bin (f59f66e649bce) [CoreOptions, JavaConfiguration,
PlatformConfiguration,
```

`(f59f66e649bce)` is a convenience hash of the rule's configuration (you can see its full value with `$ bazel config f59f66e649bce`). This output shows that `//:cc_bin` requires CppConfiguration (which is expected) and JavaConfiguration (because it depends on `//:java_bin`). It also shows that `//:java_bin`'s output doesn't require CppConfiguration.

Measuring overhead

`ctexplain` uses `cquery deps()` with `--show_config_fragments` to identify all of a build's configured targets and their transitive fragment dependencies. It uses this to report how many more *configured targets* there are than *targets* and which ones can be merged because they don't use the parts of configuration that change.

It supports two modes:

Single-build efficiency

When passed a single build command (targets and command-line build flags), `ctexplain` reports the build's *configured target* / *target* ratio, which configured targets are unnecessarily forked, and which config settings create those forks.

Example:

```
$ ctexplain //experimental:simple_split
Collecting configured targets for //experimental:simple_split... done in 0.19 s.
Collecting configurations... done in 0.20 s.
Configurations: 3
Targets: 38
Configured targets: 60 (+57.9% over target count)
Targets with multiple configs: 22 (44 of the 60 configured targets)
Configured targets that can be merged with others: 44
Total configured targets with optimal trimming: 38 (+0.0% over target count,
-36.7% vs. untrimmed graph)

Option changes that create 22 unnecessary extra CTs (60 untrimmed - 38 trimmed =
22 unneeded):
 - 22 CTs:
     options:
        "python_version": "PY2" -> "PY3"
     labels:
        1: //experimental:non_py_dep1
        2: //experimental:non_py_dep2
        ...
```

Cross-build shareability

When passed two build commands (two sets of targets and build flags), `ctexplain` reports how much their configured target graphs overlap.

This can help show how much work must be re-done when repeating a build with different flags or how much remote caching could help cross-user build speed.

Example:

```
$ ctexplain
    --build1="//experimental:buildme --define a=1"
    --build2="//experimental:buildme --define a=2"
Evaluating combined configured targets of builds:
 #1: '//experimental:buildme --define a=1'
 #2: '//experimental:buildme --define a=2'

Collecting configured targets for #1... done in 0.11 s.
Collecting configurations for #1... done in 70 ms.
Collecting configured targets for #2... done in 0.20 s.
Collecting configurations for #2... done in 65 ms.
```

```
Just #1's graph:
----------------------------------------------------------------------
 Configurations: 1
 Targets: 34
 Configured targets: 34 (+0.0% over target count)
 Targets with multiple configs: 0 (0 of the 34 configured targets)
 Configured targets that can be merged with others: 0
 Total configured targets with optimal trimming: 34 (+0.0% over target count,
-0.0% vs. untrimmed graph)

Just #2's graph:
----------------------------------------------------------------------
 Configurations: 1
 Targets: 34
 Configured targets: 34 (+0.0% over target count)
 Targets with multiple configs: 0 (0 of the 34 configured targets)
 Configured targets that can be merged with others: 0
 Total configured targets with optimal trimming: 34 (+0.0% over target count,
-0.0% vs. untrimmed graph)

Combined graph of both #1 and #2:
----------------------------------------------------------------------
 Configurations: 2
 Targets: 34
 Configured targets: 68 (+100.0% over target count)
 Targets with multiple configs: 34 (68 of the 68 configured targets)
 Configured targets that can be merged with others: 40
 Total configured targets with optimal trimming: 48 (+41.2% over target count,
-29.4% vs. untrimmed graph)

Option changes that create 20 unnecessary extra CTs (68 untrimmed - 48 trimmed =
20 unneeded):
 - 20 CTs:
     options:
        "--define:a": "1" -> "2"
     labels:
        1: //experimental:dep1
        2: //experimental:dep2
        ...
```

It's important to note that neither mode offers solutions. The core program is simply a *measuring* tool. Further analysis is needed to figure out *how* to merge configured targets, what kinds of build refactoring would help, and what changes to Bazel would help.

But this is still a tool that can grow. Even just as pure measurement, more queries can and should be added to its framework to offer more pointed guidance for these decisions.

Implementation

We'll port the existing Python program from its current private experimental location to github.com/bazelbuild/bazel/tools/ctexplain. Tests will be added to Bazel's CI pipeline.

`ctexplain`'s basic model:

1. Subprocess-call `bazel cquery --show_config_fragments=transitive` to get the build's configured targets and their transitively required fragments.

   Future work could provide a dedicated `bazel` Python API and `--show_config_fragments` results in `cquery` 's machine-readable output.
2. Collect the hashes of all configurations from step 1.
3. Subprocess-call `bazel config <configHash>` to collect each configuration's fragments and option values. Use `config`'s `--output=json` mode for easy parsing.

   Now we have a list of configured targets, which fragments they require, and which build options actually differ between configurations. This is enough information to get our final results.

   `--define` and Starlark flags are considered direct dependencies (they're not associated with broader fragments).
4. Map changed build options to the fragments they belong to. This determines which configured targets can be merged. If the same target exists in two configurations and its required fragments have the same options, these can be merged.
5. Compute and report all the results we need.

One unfortunate complication is the distinction between fragments and fragment options. They're conceptually similar (both have definitions for Java settings) but are used in different places. Rules declare which *fragments* they require, while configuration state is stored in *fragment options*. Reconciling them requires the mapping in step 4.

Worse, this is not a 1-1- mapping. `bazel config` will be extended to report the right mapping. Future work will merge these concepts as much as is practical.

The implementation includes a well-defined internal API to make coding new queries easy. The example output in this design is by no means the limit of what the tool can report.

Future enhancements

This proposal is limited to a basic framework for querying configuration efficiency and a set of useful initial queries.

As time permits, we can consider the following extensions:

- **Tool integration**

    - Machine-readable output
    - Programmatic API
    - Integration with build health dashboards
    - Visualizable graph output
- **Root cause analysis**
    - Which configured targets contribute the most bloat, as measured by cumulative size of unnecessarily forked sub-graphs?
        - These are ideal trimming candidates.
    - Which configured targets contribute the most bloat, as measured by how much they increase graph-wide fragment requirements?
        - These are ideal refactoring candidates.
        - Example: a low-level dependency with a `select()` on `--foo=bar`.
    - Which flag changes cause the most bloat?
- **Advice**
    - Show the relative benefit of trimming (keeping the same build structure but making Bazel smarter) vs. refactoring (moving around dependencies).
    - Identify candidates for dependency injection
    - Identify flag combinations that maximize reuse

Precedent

This tool was inspired by a similar tool called CTbloat, developed by Jon Brandvein.

That tool is also experimental and not publicly available.

Future work could integrate these tools.

Document History

Date	Description
2020-02-28	First proposal