Bazel

Incremental builds with small memory footprint

Please read Bazel Code of Conduct before commenting.

- **Authors**: ilist@google.com, lberki@google.com, nharmata@google.com, twerth@google.com

- **Status**: **Draft** | In review | Approved | Rejected | In progress | Implemented
- **Reviewers**: TBD
- **Created**: 2023-10-23
- **Updated**: 2023-10-23
- **Discussion thread**: <link>

Introduction

This document describes a project that could very well result in cutting the retained heap use of Bazel to a fraction of its current amount for interactive use cases with only minor compromises in usability and which could be prototyped reasonably quickly.

Problem statement

Bazel, as it is, maintains the dependency graph of the whole build in RAM. This, coupled with the fact that the Google Way is to check in as few prebuilt binaries as possible and to build everything demand results in Bazel having to maintain the dependency graph for a the whole transitive closure of the code it builds, from the compiler through base and intermediate libraries to the actual code one is interested in.

This is wasteful because most of the time, any user only changes a vanishingly small fraction of all the source files that are needed to build the code they are interested in. Yet, Bazel spends a large amount of resources maintaining the dependency graph so that it can handle changes to any source file.

Design

We propose to add the concept of "working sets" to Bazel: the set of files the user is likely to change, explicitly specified by them or the IDE they are using. This typically means the source code of a single library or binary and that of the associated tests.

Then, after an initial build, Bazel would classify Skyframe nodes into the following classes:

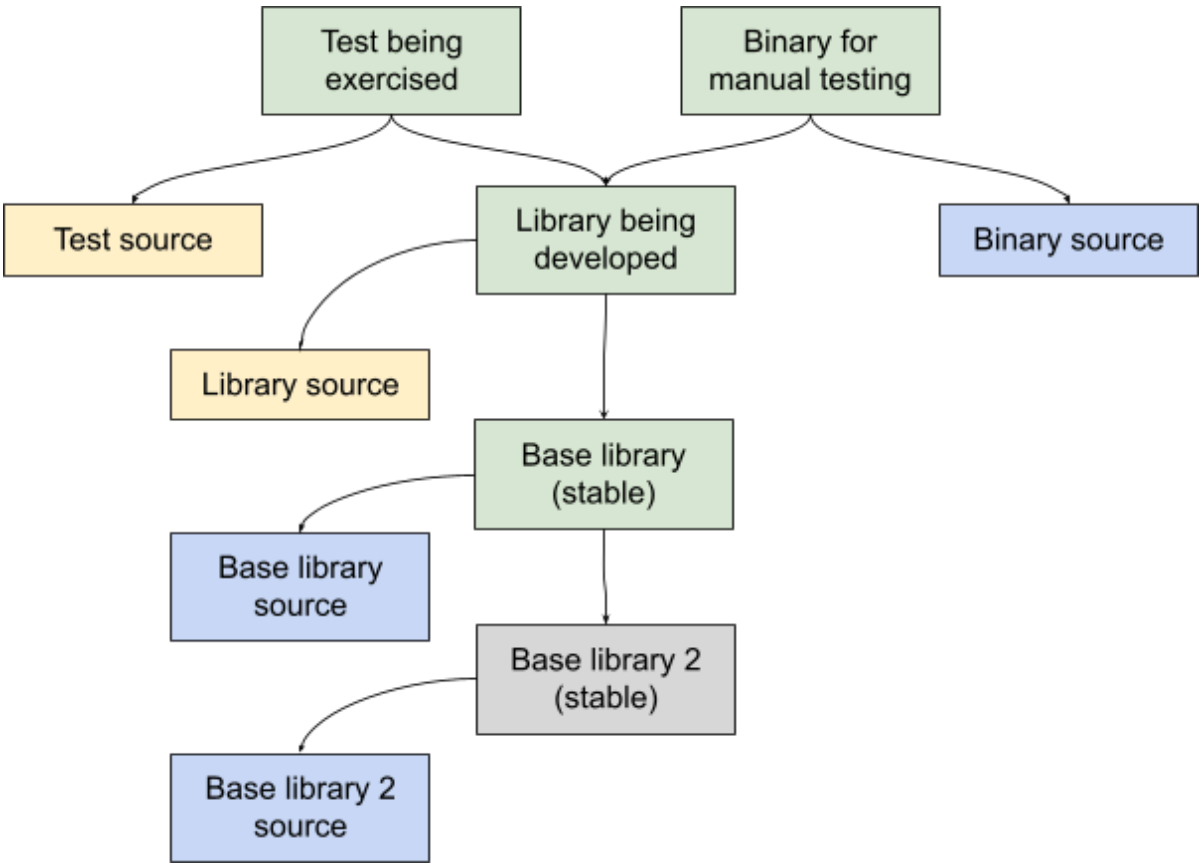
1. **Nodes that transitively depend on the working set.** These are needed for incremental builds if only the working set changes, so they need to be kept.
2. **Direct dependencies of the nodes in (1).** These are also needed for incremental builds if only the working set changes, thus, they are also kept.
3. **Other non-leaf nodes.** These are not necessary to handle changes in the working set, so they are discarded.
4. **Other leaf nodes** (for example, source files or configuration flags). These are needed to know whether we need a node in (3) for incremental builds. Thus, these are kept in a flat set, discarding the graph structure.

Then, on each subsequent Bazel invocation, one of the two things can happen:

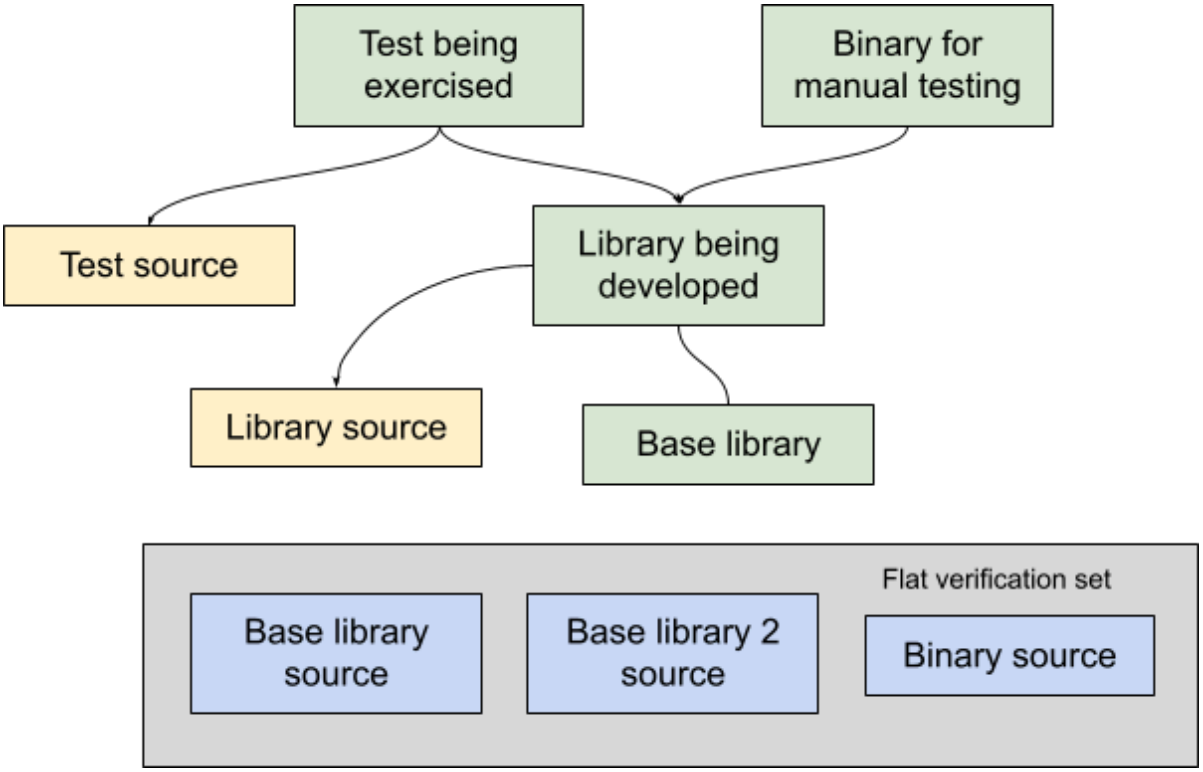
- **Happy path:** only nodes in the declared "working set" are invalidated. In this case, Bazel can do its job based solely on the minimized Skyframe graph because the discarded nodes would provably not change.
- **Unhappy path:** at least one leaf node changes which invalidates discarded nodes, the command line flags change or a new target is added to the command line. In this case, a full re-analysis needs to happen and thus the Bazel invocation would both use much more RAM and be much slower, comparable to a cold Bazel invocation. This is expected to happen infrequently if the working set is chosen well.

In graphs, if the original Skyframe graph looks like this with the color coding below:

- Yellow: sources in working set
- Blue: sources not in working set
- Gray: Discarded nodes
- Green: Kept nodes



The graph would look like this after discarding unnecessary nodes:



Note that the structure of the unnecessary part of the graph and its connection to the live nodes are also discarded. This, in practice, means that the following data structures can all be discarded from there:

- Actions
- Configured targets and aspects
- Packages (and targets within them)
- Java / Starlark data structures that are not referenced from live nodes (including e.g. CcCompilationContext, but not the individual nested sets in it)

The only thing that would be kept from discarded nodes are the data structures that have an incoming reference from the Java object graph generated by the live nodes (for example, nested sets)

Benefits

The peak post-analysis and post-build heap usage would not change; the initial build would be much the same as it is today. However, the heap use of subsequent incremental builds would shrink by a lot. It's difficult to estimate how much, but it would be surprising if this proposal would not reduce it by at least a factor of five.

This would have a number of beneficial effects:

- The age-old "Bazel is eating all the RAM of my workstation" would go away
- It would be more feasible to keep multiple Bazel instances in RAM for longer

Working sets are already a known concept in IDEs (e.g. the IntelliJ plugin) so users would require minimal education.

Costs

This would be a significant change to how Skyframe and Bazel work. However, the change is mostly localized to Skyframe and it would be possible to gauge the potential impact with a cheap prototype, thereby limiting the risk.

Unlike Skymeld, the new mode of operation would stay optional forever, so this project is not subject to the same risk as Skymeld, which must work in every use case Bazel has and thus needs to conform to the legacy behavior of Bazel much more closely.

Time estimates:

- **Barely functional prototype:** ~two weeks of focus time if done by an engineer well versed in Skyframe. This will answer whether the concept is feasible, provide a good estimate on the expected memory savings and tell what unexpected complications will need to be resolved.
- **Dogfoodable implementation:** ~a quarter of engineering time (but this estimate is very rough; it would be possible to give a better estimate after the prototype is complete)

Risks

- The memory benefit is not as much as we expect (the prototype would answer this question)
- The implementation ends up being complex enough so as to be unsupportable (the prototype would remove this risk)

Further improvements

If this change is successful, it would synergize well with a number of other changes that would be useful for Bazel for different reasons.

Serializing state

If we were able to serialize the minimized Skyframe graph to a reasonable number of bytes, it would unlock a number of use cases:

- Distributing Skyframe states between developers working on the same team from a source state at a known green change (thereby making it possible to forgo the first "full analysis" invocation required), thereby making the cost of starting to work in a new source tree much smaller.
- A developer could keep multiple serialized states around, thereby allowing them to flip the value of a command line flag without incurring a costly re-analysis.

Limiting scope of command line flags

If we made it possible to limit how far a particular command line flag (e.g. --copt) is propagated in the configured target graph, it would be possible to change the value of that flag without a costly full re-analysis.