Bazel C/C++ Transitive Libraries - Attack Plan

This is a publicly readable document!

Author: hlopko@google.com
Status: Being reviewed
Reviewers: lberki@google.com, ulfjack@google.com, dslomov@google.com, klimek@google.com, schmitt@google.com
Date: July 2017

This is a distilled version from the original design doc containing only parts that are going to be implemented.

Introduction

In Google, we build binaries that link all their dependencies statically. Therefore this use case is well settled, polished, and optimized in Bazel. In the outside world however, people need greater control over which libraries get linked statically and which dynamically (#492), as well as which objects the library is linked against.

Bazel cannot yet create a static archive that contains all objects from all its transitive dependencies. This has been asked for internally and externally (#1920, #2253). For this reason Tensorflow currently cannot distribute static library, only .so (#5511).

During the linking of the binary, the program linker throws away all the symbols that are not transitively used in the .o files of that binary. This includes the removal of global variables whose initializers are used to register functionality of the library in global registries, unless another symbol from the translation unit that defines them is (transitively) used from the binary.

Bazel can create a dynamic library that contains all its transitive dependencies, but it lacks the mechanism to control which dependencies should be linked in and which shouldn't (specifically, we cannot leave some symbols undefined), making it susceptible to the diamond linking problem (getting static initialization of a single library present in the resulting binary or library multiple times).

Requirements

1. To be able to build transitive static library as an artifact ready for distribution

2. To be able to build transitive dynamic library as an artifact ready for distribution
3. cc_binary should be able to depend on (transitive|nontransitive) dynamic library and on transitive static library (as it is useful to detect link ordering surprises).
4. cc_library should be able to depend on transitive dynamic library. This will allow users to prevent ODR violation by packaging their libraries cleanly.
5. Library artifact ready for distribution can be one of:
    a. Single static library containing all statically reachable transitive objects. In case a static library is a dependency of this transitive library, bazel will merge these libraries.
    b. Single merged dynamic library with all statically reachable transitive objects
    c. Single merged static library with ‘a 'solib' directory with dynamic libraries that the lib depends on
    d. Single merged dynamic library with a 'solib' directory with dynamic libraries that the lib depends on

Attack Plan

We will introduce new dependency type to cc_library and friends: ‘reexport_deps’. As implicit outputs of cc_library rules are an implementation detail and users should not depend on them, we will introduce 'cc_static_library' and 'cc_shared_library' rules to make the library generation explicit. These rules have no srcs and deps attributes - they only have 'reexport_deps'. Other cc_library rules can depend on cc_static_library and cc_shared_library.

cc_static_library and cc_shared_library

Right now binary artifacts in the C++ world are cc_binary and cc_test. cc_binary is used to build executable binaries, and transitive shared libraries linking all its transitive dependencies, but without a way of excluding some (that's one of the reasons why this design doc exists). As a part of rolling out transitive libraries described by this design doc producing shared libraries from cc_binary will be deprecated. In other words, there will be no supported way of creating static or shared library that contains dependencies from 'deps', only 'reexport_deps' will be linked against/archived. We hope to keep it this way.

cc_static_library and cc_shared_library will have no srcs or deps (that is what cc_library is for) and will only have reexport_deps dependency type. They will unconditionally produce library artifacts of their respective type.

Reexport_deps

These dependencies will be linked into the output artifacts of the parent cc_*_library target. This doesn't work transitively. If you need transitive behaviour, you need to mark all the dependencies down the road as reexport_deps.

Examples

For the sake of brevity I omit the top_level cc_static_library or cc_shared_library rules from all the cases except Case 0.

Case 0: cc_*_library rules usage

```
cc_static_library(
  name = 'main_staticlib',
  reexport_deps = [ ':main' ]
)
cc_shared_library(
  name = 'main_sharedlib',
  reexport_deps = [':main']
)
cc_library(
  name = 'main',
  srcs = [ 'main.cc' ],
  deps = [ ':base' ],
)
cc_library(
  name = 'base',
  srcs = [ 'base.cc' ]
)
```

Case 1: cc_library reexporting cc_library

```
cc_library(
  name = 'main',
  srcs = [ 'main.cc' ],
  reexport_deps = [ ':base' ],
)
cc_library(
  name = 'base',
  srcs = [ 'base.cc' ]
)
```

Observable effect (comparing using reexport_deps over deps):
Archiving of //:'main_staticlib' will turn from:

```
ar rcsD libmain.a main.o
```

into:

```
ar rcsD libmain.a main.o base.o
```

Creating 'main_sharedlib' library will turn from:

```
clang -shared -o libmain.so main.o
```

into:

```
clang -shared -o libmain.so main.o base.o
```

Case 2: cc_binary reexporting cc_library

This has no observable effect. cc_binary will by default use all transitive libraries, and add all non-reexport deps. At the end, the set of objects linked into cc_binary is the same no matter how you use reexport_deps.

Case 3: Reexporting a dep that reexports a dep.

```
cc_library(
  name = 'top',
  srcs = [ 'top.cc' ],
  reexport_deps = [ ':middle' ],
)
cc_library(
  name = 'middle',
  srcs = [ 'middle.cc' ],
  deps = [ ':sideways' ],
  reexport_deps = [ ':bottom' ],
)
cc_library(
  name = 'bottom',
  srcs = [ 'bottom.cc' ],
)
cc_library(
  name = 'sideways',
  srcs = [ 'sideways.cc' ],
)
cc_binary(
  name = 'binary',
  deps = [ ':top' ],
  srcs = [ 'binary.cc' ],
)
```

Observable effect (comparing using reexport_deps over deps):
(green denotes flags added, red flags removed)

Archiving:

```
ar rcsD libbottom.a bottom.o
ar rcsD libsideways.a sideways.o
ar rcsD libmiddle.a middle.o bottom.o
ar rcsD libtop.a top.o middle.o bottom.o
```

Linking shared library:

```
clang -shared -o libbottom.so bottom.o
clang -shared -o libsideways.so sideways.o
clang -shared -o libmiddle.so middle.o bottom.o
clang -shared -o libtop.so top.o middle.o bottom.o
```

Linking binary:

```
clang -o binary binary.o -ltop -lmiddle -lsideways -lbottom
```

Case 4: Reexporting a dep that reexports precompiled static library

```
cc_library(
  name = 'top',
  srcs = [ 'top.cc' ],
  reexport_deps = [ ':middle' ],
)
cc_library(
  name = 'middle',
  srcs = [ 'middle.cc' ],
  reexport_deps = [ ':precompiled_static_lib' ],
)
cc_library(
  name = 'precompiled_static_lib',
  srcs = [ 'static_lib.a' ],
)
cc_binary(
  name = 'binary',
  deps = [ ':top' ],
  srcs = [ 'binary.cc' ],
)
```

Observable behaviour (comparing using reexport_deps over deps):
(green denotes flags added, red flags removed)

This cannot be done using ar in a normal way (on windows link.exe supports this, on mac libtool supports this), we would need to use MRI script. The ar invocation becomes simple `'ar -M < script'`, where 'script' contains following:

Archiving middle

```
create libmiddle.a
addmod middle.o
addlib precompiled_static_lib.a
save
end
```

Archiving top

```
create libtop.a
addmod top.o
addmod middle.o
addlib precompiled_static_lib.a
save
end
```

Linking shared library:

```
clang -shared -o libtop.so top.o middle.o -lprecompiled_static_lib
clang -shared -o libmiddle.so middle.o -lprecompiled_static_lib
```

Linking binary:

```
clang -o binary binary.o -ltop ~~-lmiddle -lstatic_lib~~
```

Case 5: Reexporting precompiled shared library

```
cc_library(
  name = 'top',
  srcs = [ 'top.cc' ],
  reexport_deps = [ ':precompiled_shared_lib' ],
)
cc_library(
  name = 'precompiled_shared_lib',
  srcs = [ 'shared_lib.so' ],
)
cc_binary(
  name = 'binary',
  deps = [ ':top' ],
  srcs = [ 'binary.cc' ],
)
```

Merging shared libraries is not possible (on neither linux, mac, nor windows). the only thing we can do is to provide the shared library in the runfiles.

Archiving is not affected, except that the shared library is distributed along the static library.
Linking shared library (again, libshared_lib.so must be distributed in the runfiles):

```
clang -shared -o libtop.so top.o -lshared_lib
```

Linking binary (with libtop.so and libshared_lib.so in the runfiles):

```
clang -o binary binary.o -ltop -lshared_lib
```

Non cc rules

There are many non C++ related rules that cc_rules can depend on, that could in theory participate in reexporting deps. The most prominent are proto_library and go_library.

proto_library

To make protos work transitively, we propose we add a boolean flag to cc_proto_library to treat all nested protos as reexport_deps, plus a boolean flag stating whether to link against/archive the proto runtime too.

go_library

When linking a binary, go linker needs to know about all of transitive go_library dependencies in order to generate runtime structures correctly. Because of this we assume people don't want to distribute shared libraries that include go runtime very often, as such dependencies couldn't be used if some other shared library included go runtime too. If I'm proved wrong, we could introduce cc_export_deps into go_library and implement all the machinery around it in the go rules.

Detecting diamonds

Bazel can automatically detect diamonds and report errors when some is discovered (with the ability to turn this check off if an user wants to keep their diamonds). The problem is that the best algorithm so far grows linearly in the number of transitive dependencies for every cc_binary, cc_shared_library, and cc_test in the graph, therefore grows quadratically in total. This is a very restrictive property of the approach if done in the analysis phase. But we can do this check in the execution phase, possibly distributing the work over multiple machines. The work is still done though, so we will keep a close eye on the performance and if this is causing trouble we will investigate possible solutions. And if we solve it, we will publish a paper about the solution :)

Case study: static initialization logic (the diamond)

Sometimes cc_libraries contain some static initialization logic that shouldn't happen twice. In the following situation:

```
cc_binary(
  name = 'main',
  srcs = [ 'main.cc' ],
  deps = [ ':leftlib', ':rightlib' ],
)
cc_shared_library(
  name = 'leftlib',
  reexport_deps = [':left']
)
cc_shared_library(
  name = 'rightlib',
  reexport_deps = [':right']
)
cc_library(
  name = 'left',
  srcs = [ 'left.cc' ],
  reexport_deps = [ ':base' ],
)
cc_library(
  name = 'right',
  srcs = [ 'right.cc' ],
  reexport_deps = [ ':base' ],
)
cc_library(
  name = 'base',
  srcs = [ 'i_have_static_initialization.cc' ],
)
```

If we are unlucky, and build 'leftlib' and 'rightlib', it might happen that the static initialization from 'base' will be inlined into their 'init' sections, and when loaded by 'main', the static initialization will happen twice. (read more in the original transitive libraries design doc). To give users control over this sensitive behavior, we advise leaving such dependencies in the 'deps'. cc_binary will link against all non-reexport deps, or creating a separate cc_shared_library for 'base' and linking against that.
To say it once more: there is no way of creating shared library that will link against dependencies in 'deps'!

Related Work

I talked about this problem with many people, and in the process we came up with two more dependency types that might help with expressivity of our C++ rules. After thinking deeply, I don't think they're needed. In particular these were:

initialization_deps - dependencies containing static initialization logic as elaborated in the case study above. There's no need to explicitly mark these I think. The only use I can think of right

now is that they would give us the ability to build a shared library containing only the initialization logic.

alwayslink_deps - these dependencies will be linked in the whole archive block (all their symbols will be present in the library, even when they're not used). This dependency is still useful, but not in the context of transitive libraries, so I wouldn't tie it's implementation with transitive libraries

Also, this document mentions depending on prebuilt libraries. This is currently very cumbersome in Bazel, one cannot wrap these libraries in whole archive block, they cannot form a lib group, etc. **cc_*_library_import** rules are solution to these issues. Transitive libraries are not affected by that work though.

Future Work

Dead code elimination

With every use of whole-archive we increase the size of the artifact. Symbols that are not used are kept around, retaining possibly dead code. This overhead is significant for e.g. mobile development. Right now, when bazel builds shared library it links all transitive objects in a whole-archive block. With transitive libraries approach this doesn't have to be the case, but this is yet to be investigated and designed. I don't think the approach described in this doc will make this work any harder.

Symbol visibility specification (as entry points)

Related to the previous problem, right now Bazel doesn't provide a way of specifying which symbols should be kept in the resulting libraries. This problem becomes more important on windows, where this information is required when building a .dll. Again, I don't think the approach described in this doc makes this work any harder.

Exposed header specification

This also needs more investigation. Since it's tied to the transitive libraries, it might be worthwhile discussing this here. Should headers of my reexported deps be available to my dependants? Should this be configurable?