Importing precompiled C++ libraries

This is a publicly readable document!

Author: hlopko@google.com
Status: being implemented
Last change: 2017-11-16 11:50
Reviewers: lberki@google.com, ulfjack@google.com, dslomov@google.com, klimek@google.com, pcloudy@google.com
go/bazel-importing-precompiled-cpp-libraries

Problem Description

Many projects (especially external ones) have dependencies that are best dependent upon using their distribution artifacts. Those are either static or shared libraries. Currently these dependencies are specified in cc_library.srcs, but this approach has many limitations:

1. Bazel has to inspect the extension in order to decide whether the file is a static or shared library. This means that bazel needs to know what extensions are common on various operating systems.
2. On Windows, shared library consists of .dll with the actual code, and .lib which serves as an interface library. Linker links against .lib file, but .dll is loaded at runtime. This cannot be expressed using existing mechanism and is blocking windows adoption. (To add confusion, static libraries are also .lib files -- same as interface libraries but with implementations.)
3. Right now, the only way of specifying that a particular static library should be linked as whole-archive is to change its extension to .lo. Bazel creates .lo file as a whole-archive static library from cc_library rules and uses it as such. This behavior is not well documented, is platform-dependent, and shouldn't be relied on. Second problem is that currently, cc_library with alwayslink = 1 will not wrap precompiled libraries in whole-archive, alwayslink only affects the output of cc_library, not its sources.
4. There is no way of specifying that a group of static libraries forms a lib-group ~~(the group of archives that are repeatedly searched until no undefined symbols are discovered)~~. I assume in practice libgroups are only needed for precompiled libraries and not cc_libraries in general, since in the more than 10 years of google-internal Bazel existence nobody needed this badly enough to be implemented. We do have a feature request for libgroups for precompiled libraries though.

Related Issues

https://github.com/bazelbuild/bazel/issues/3402 - tracking issue for this work
https://github.com/bazelbuild/bazel/issues/3323 - cannot link against extension-less library, this will be fully fixed by this work.
https://github.com/bazelbuild/bazel/issues/2944 - cannot link imported archive as whole-archive, this will be fully fixed by this work.
https://github.com/bazelbuild/bazel/issues/818 - is only slightly related but worth mentioning here. The problem is there is no way of expressing lib groups for cc_libraries, something we discuss in #4 in Problem Description. This will not be solved by this work.
https://github.com/bazelbuild/bazel/issues/1534 - Cannot link to library whose basename is not the same as its soname. This work will not solve this, but at least will provide a principled place where such behavior should be implemented.

Proposed Solution

I propose we introduce 1 new rule, 'cc_import'. Attributes of cc_import:

- static_library: single static library
- alwayslink: true when the 'static_library' should be inside the whole-archive block
- hdrs: collection of include headers
- shared_library: single precompiled shared library
- interface_library: interface library corresponding to 'shared_library' (mostly this is only there to make Windows dlls work, we can create elf interface libraries internally, but externally this is not common)
- system_provided: true when we expect the shared library is provided by the system during runtime.
- includes (and other related attributes)

It enables us to eventually remove bits of file detection logic from cc_library implementation, and brings us closer to the desired state where cc_library is just a simple collection of sources and headers (or a collection of objects and related stuff if looking from the other side). I also expect more features to be added in the future such as 'this is the shared library you should use, but expect it will be installed somewhere else when the binary runs'.

This solution doesn't solve problem #4 (lib groups). Based on the discussion with experts (Appendix 1), I think providing lib groups only for precompiled libraries will not be enough for all the known use cases. I expect separate rule (aka 'cc_lib_group') will be needed.

Alternative Solutions

Specific attributes (annotating edges, not nodes)

We could introduce 'static_lib_deps' and 'shared_lib_deps' attributes. This solves the problem #1. #2 can be solved by expecting "magic" prefix or suffix (e.g. 'interface') for interface libraries corresponding to shared libraries entered in the 'shared_lib_deps'. #3 can be solved by modifying how alwayslink behaves. Instead of only affecting the library artifact created from cc_library it can also affect all declared static-lib-deps. #4 could be solved by a) introducing an explicit rule (e.g. cc_lib_group) or b) adding another attribute such as 'libgroup' = 1 to cc_library.

I argue cc_import rule is superior because they simplify the cc_library implementation and are more concise. Attributes specific only to static_lib_deps, or shared_lib_deps pollute the (otherwise already very polluted :) set of attributes of cc_library.

Multiple cc_import rules

We could split the cc_import rules into two (e.g. cc_static_library_import, cc_shared_library_import). We could allow multiple archives to be specified in a single cc_static_library_import and add a boolean attribute specifying whether archives should form a lib-group. As already mentioned, this would not solve all the lib-group use cases we know about. This approach is also more difficult to use in scenarios where users want to use both static and shared libraries depending on the linkstaticness of the top level cc_binary rule.

Appendix 1 - lib group use cases

Tim Blakely:
I only have an anecdotal example but the reason I was interested in start-/end-group support is that it is used in a few microcontroller SDKs that use circular dependencies during linking; namely the ESP8266 SDK and ESP32 SDK. It's been a while since I've toyed with it, so I don't know whether the circular dependencies were introduced at the chip level SDKs from Espressif or further down the software stack at the Xtensa core SDK level. Either way, the circular dependencies are not at a level that's possible to change without rewriting the entire SDK. A custom CROSSTOOL chain with a start-/end-group flag_set got around this, so it's not blocking by any means.

Austin Schuh:
For one of my projects, I've got to build the standard C and C++ libraries for a microcontroller. We are bug-fixing the standard C library enough that it made sense to spend a couple weeks and pull it into bazel. Proper support would have saved me a couple days of pain, but it's done now.

I'm currently working around it by re-implementing the entire cc_binary and cc_library rules (so we can support multiple architectures in the same package), extracting the .a library files in a genrule, and then combining all of the .a's in a group back into a single .a with ar directly. (libstdc++ has circular internal dependencies, along with newlib. Sigh...) Long term when dynamic configurations arrive, we should be able to drop our custom rules and will then need this support.

James Y Knight:
Source files within a single library are often circularly-dependent, and that's generally accepted to be fine. The default behavior when searching an archive is to search all the object files within it (no ordering requirement), and thus users don't typically do anything special to get the desired linker behavior.

However, in bazel, this can be problematic, because copts can only be specified for an entire "cc_library" rule at a time. If you want to specify different compilation options for different source files within a single logical library (e.g. have some sources built with copts = ["-O3"] and some not), you must split into multiple cc_library rules. But, once you do that, you've lost the implicit circular-dependency allowance, and there's now no way to express the dependencies. Of course, you didn't really want these object files to end up in two separate archives in the first place, but every cc_library turns into a separate archive whether you like it or not, and now you're stuck.

Currently, people resort to using alwayslink=1 to workaround the issue, which is not ideal.

I was just reminded of another current workaround for this issue which is hardcoded in blaze: the objc_library "non_arc_srcs" attribute. I don't know the history of this attribute, so I don't know if it was introduced for this reason or not. But at this point, the only thing it provides is the ability to pass the "-fno-objc-arc" to the compilation of some objects in the library and not others -- an ability which isn't generally available.