Linker-plugin-LTO

The `-C linker-plugin-lto` flag allows for deferring the LTO optimization to the actual linking step, which in turn allows for performing interprocedural optimizations across programming language boundaries if all the object files being linked were created by LLVM based toolchains. The prime example here would be linking Rust code together with Clang-compiled C/C++ code.

Usage

There are two main cases how linker plugin based LTO can be used:

- compiling a Rust `staticlib` that is used as a C ABI dependency
- compiling a Rust binary where `rustc` invokes the linker

In both cases the Rust code has to be compiled with `-C linker-plugin-lto` and the C/C++ code with `-flto` or `-flto=thin` so that object files are emitted as LLVM bitcode.

Rust `staticlib` as dependency in C/C++ program

In this case the Rust compiler just has to make sure that the object files in the `staticlib` are in the right format. For linking, a linker with the LLVM plugin must be used (e.g. LLD).

Using `rustc` directly:

```
# Compile the Rust staticlib
rustc --crate-type=staticlib -Clinker-plugin-lto -Copt-level=2 ./lib.rs
# Compile the C code with `-flto=thin`
clang -c -O2 -flto=thin -o main.o ./main.c
# Link everything, making sure that we use an appropriate linker
clang -flto=thin -fuse-ld=lld -L . -l"name-of-your-rust-lib" -o main -O2 ./cmain.o
```

Using `cargo`:

```
# Compile the Rust staticlib
RUSTFLAGS="-Clinker-plugin-lto" cargo build --release
# Compile the C code with `-flto=thin`
clang -c -O2 -flto=thin -o main.o ./main.c
# Link everything, making sure that we use an appropriate linker
clang -flto=thin -fuse-ld=lld -L . -l"name-of-your-rust-lib" -o main -O2 ./cmain.o
```

C/C++ code as a dependency in Rust

In this case the linker will be invoked by `rustc`. We again have to make sure that an appropriate linker is used.

Using `rustc` directly:

```
# Compile C code with `-flto`
clang ./clib.c -flto=thin -c -o ./clib.o -O2
# Create a static library from the C code
ar crus ./libxyz.a ./clib.o
```

```
# Invoke `rustc` with the additional arguments
rustc -Clinker-plugin-lto -L. -Copt-level=2 -Clinker=clang -Clink-arg=-fuse-ld=lld
./main.rs
```

Using `cargo` directly:

```
# Compile C code with `-flto`
clang ./clib.c -flto=thin -c -o ./clib.o -O2
# Create a static library from the C code
ar crus ./libxyz.a ./clib.o

# Set the linking arguments via RUSTFLAGS
RUSTFLAGS="-Clinker-plugin-lto -Clinker=clang -Clink-arg=-fuse-ld=lld" cargo build -
-release
```

Explicitly specifying the linker plugin to be used by `rustc`

If one wants to use a linker other than LLD, the LLVM linker plugin has to be specified explicitly. Otherwise the linker cannot read the object files. The path to the plugin is passed as an argument to the `-Clinker-plugin-lto` option:

```
rustc -Clinker-plugin-lto="/path/to/LLVMgold.so" -L. -Copt-level=2 ./main.rs
```

Usage with clang-cl and x86_64-pc-windows-msvc

Cross language LTO can be used with the x86_64-pc-windows-msvc target, but this requires using the clang-cl compiler instead of the MSVC cl.exe included with Visual Studio Build Tools, and linking with lld-link. Both clang-cl and lld-link can be downloaded from LLVM's download page. Note that most crates in the ecosystem are likely to assume you are using cl.exe if using this target and that some things, like for example vcpkg, don't work very well with clang-cl.

You will want to make sure your rust major LLVM version matches your installed LLVM tooling version, otherwise it is likely you will get linker errors:

```
rustc -V --verbose
clang-cl --version
```

If you are compiling any proc-macros, you will get this error:

```
error: Linker plugin based LTO is not supported together with `-C prefer-dynamic`
when
targeting Windows-like targets
```

This is fixed if you explicitly set the target, for example `cargo build --target x86_64-pc-windows-msvc` Without an explicit --target the flags will be passed to all compiler invocations (including build scripts and proc macros), see cargo docs on rustflags

If you have dependencies using the `cc` crate, you will need to set these environment variables:

```
set CC=clang-cl
set CXX=clang-cl
set CFLAGS=/clang:-flto=thin /clang:-fuse-ld=lld-link
set CXXFLAGS=/clang:-flto=thin /clang:-fuse-ld=lld-link
REM Needed because msvc's lib.exe crashes on LLVM LTO .obj files
set AR=llvm-lib
```

If you are specifying lld-link as your linker by setting `linker = "lld-link.exe"` in your cargo config, you may run into issues with some crates that compile code with separate cargo invocations. You should be able to get around this problem by setting `-Clinker=lld-link` in RUSTFLAGS

Toolchain Compatibility

In order for this kind of LTO to work, the LLVM linker plugin must be able to handle the LLVM bitcode produced by both `rustc` and `clang`.

Best results are achieved by using a `rustc` and `clang` that are based on the exact same version of LLVM. One can use `rustc -vV` in order to view the LLVM used by a given `rustc` version. Note that the version number given here is only an approximation as Rust sometimes uses unstable revisions of LLVM. However, the approximation is usually reliable.

The following table shows known good combinations of toolchain versions.

Rust Version	Clang Version
Rust 1.34	Clang 8
Rust 1.35	Clang 8
Rust 1.36	Clang 8
Rust 1.37	Clang 8
Rust 1.38	Clang 9
Rust 1.39	Clang 9
Rust 1.40	Clang 9
Rust 1.41	Clang 9
Rust 1.42	Clang 9
Rust 1.43	Clang 9
Rust 1.44	Clang 9
Rust 1.45	Clang 10
Rust 1.46	Clang 10

Note that the compatibility policy for this feature might change in the future.