Advanced Compiler Usage

2024 Fall ECE454: Computer Systems Programming
Jon Eyolfson

Lecture 6
1.0.0

There’s More than One Compiler

Clang is another compiler that uses LLVM as an optimizer

LLVM IR is in SSA (single static assignment) form, and is quite readable

Let’s try it out with last lectures code!

Using Clang with Build Systems

Pretty much every build system the environment variable `CC` specifies the C compiler

You can set it for just a command:

```
CC=clang meson setup build
meson compile -C build
```

Note: `CXX` is the C++ compiler

Clang Recognizes Vectorization, and Memory Usage

Benchmark	CPE (GCC)	CPE (Clang)
combine1g	20.24	23.24
combine1	10.01	10.01
combine2	7.04	7.04
combine3	1.76	0.54
combine3w	1.76	0.53
combine4	1.54	0.53
combine4p	1.53	0.55
combine4u	1.53	0.53
combine5u2	1.18	0.57
combine5u3	1.10	1.08
combine5u4	0.65	0.75
combine5u5	1.07	0.73
combine5u8	0.55	0.62
combine5u16	0.54	0.57
combine6	0.82	0.62

Link-time Optimization (LTO)

This helps the compiler optimize at more global level

It keeps the code at an IR level for optimization, until the final linking step

The compiler doesn't have to assume the worst for functions you write in other source files

The compiler generates machine code as last as possible

Using LTO with Meson

Messing with the flags is a bit of a pain with other build tools

To use LTO, set the following:

```
meson setup -Db_lto=true build
meson compile -C build
```

Let’s see what our example looks like with LTO enabled

Performance Numbers using LTO

Benchmark	CPE
combine1g	20.21
combine1	1.76
combine2	1.76
combine3	1.76
combine3w	1.76
combine4	1.54
combine4p	1.52
combine4u	1.15
combine5u2	1.18
combine5u3	1.10
combine5u4	0.65
combine5u5	1.07
combine5u8	0.54
combine5u16	0.55
combine6	0.80

Clang also allows you to use LTO

Clang is Much More Aggressive with LTO

Benchmark	CPE
combine1g	21.29
combine1	1.56
combine2	1.57
combine3	0.12
combine3w	0.12
combine4	0.00
combine4p	0.00
combine4u	0.00
combine5u2	0.06
combine5u3	0.00
combine5u4	0.00
combine5u5	0.00
combine5u8	0.00
combine5u16	0.00
combine6	0.00

You Should Prefer to Use LTO

It takes more time and more memory than traditional builds
(LLVM also supports thinLTO for large builds that run out of memory)

Also, things can go very poorly if you mess with linking, or lie to the compiler

Since, as we saw, it can remove a lot of code, you should
absolutely not use it for development / debugging

Profile Guided Optimization

Your compiler can also use profiling data to generate better code

It's especially useful for knowing how to organize code blocks in memory

Thankfully, it's also an option for Meson, `b_lto`,
the options are: `off`, `generate`, and `use`

First, we need to compile with generate:

```
meson setup -Db_pgo=generate build
```

Generating and Using Profiling Data

It's best to include benchmarks for your project
  Your benchmarks should be representative workloads

After, we can compile with generate, benchmark, and re-compile:

```
cd build
ninja                        # Compile
ninja benchmark              # Run all the benchmarks
meson configure -Db_pgo=use  # Change setting to use the profile data
ninja                        # Re-compile
```

Just like running with `gcov`, it'll generate `.gcda` files

Note: don't be like me, your program must call `exit` to generate `.gcda` files

Performance Numbers using LTO+PGO (GCC)

Benchmark	CPE
combine1g	20.41
combine1	1.76
combine2	1.76
combine3	1.76
combine3w	1.76
combine4	0.72
combine4p	0.72
combine4u	0.65
combine5u2	0.71
combine5u3	0.54
combine5u4	0.64
combine5u5	0.53
combine5u8	0.54
combine5u16	0.54
combine6	0.69

LTO and PGO May Save You a Lot of Manual Effort

It gives your compiler much more information it can use

You'll likely have better performance gains from applying domain-specific knowledge (the fastest computation is a skipped one)

Again, your compiler will not pick data structures or data types for you
This is also where you'll get a lot of your performance