SPECTRAL:
TECHNOLOGIES

WEEK 9

Optimizing C++ programs

SPECTRAL:
TECHNOLOGIES

Plan

– Environment setup (Linux)
– **C++ features and corners**
– Utilizing CPU and RAM properly

SPECTRAL:
TECHNOLOGIES

C++

- Building flags
- std (we've covered this topic mostly)
- Hints for compiler
- Common cases

Building optimized C++ programs

- `-O3` obviously
- `-march` to build a program for specific architecture (allows to use custom instructions, like simd and etc.)
  - `-march=native` if you're on a target architecture already
- `-ffast-math` for programs with a lot of fp arithmetics
- There are a lot of flags considering inlining, e.g. `-inline-threshold=...`, `-mbranches-within-32B-boundaries`, etc
- Profile-guided optimization (PGO)
- Link-time optimization (LTO)

PGO

- Build a program with statistics collection (e.g. how often each function was executed)
- Run it and collect stats
- Recompile program using these stats

```
> clang++ -O3 -fprofile-instr-generate code.cc -o code
> LLVM_PROFILE_FILE="default.profraw" ./code
> llvm-profdata merge -output=code.profdata *.profraw
> clang++ -O3 -fprofile-instr-use=code.profdata code.cc -o code
```

SPECTRAL:
TECHNOLOGIES

PGO

- Allows to optimize your program for specific workload, can greatly improve the performance
- You can manually tune prof-files collected by PGO manually as well
- Its possible to build clang itself with PGO on your codebase and get about 10-20% faster compilation
- https://llvm.org/docs/HowToBuildWithPGO.html

SPECTRAL:
TECHNOLOGIES

LTO

- Recall, that by default each translation unit is compiled and optimized separately. Then linker merges the results together
  - Intermodular optimizations are not possible
- With link-time optimization each translation unit is compiled to intermediate bitcode
- The linker then uses libLTO and these bitcode files to optimize and generate the resulting library/executable
- Linkage obviously takes more time

LTO

- To use, compile and link everything with `-flto` flag
- There is also `-flto=thin` version, which compiles faster, but generates less optimized code in general

SPECTRAL:
TECHNOLOGIES

BOLT

- A separate tool built by facebook
- Processes an already built binary (needs to be linked with a specific flags)
- Optimizes the layout of instructions and data
- https://github.com/llvm/llvm-project/tree/main/bolt

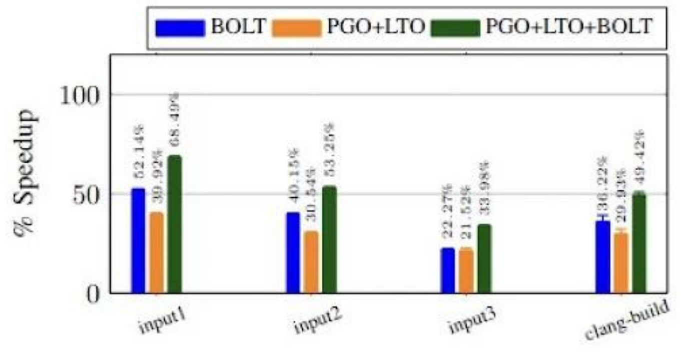

Fig. 6: Performance improvements for Clang.

Hints for compiler: branches

- Specify which branch is more likely to be executed (since c++20)

```
if (b) [[likely]] { // Placement on the first statement in the branch.
  // The compiler will optimize to execute the code here.
} else {
}
```

- On earlier versions

```
#define unlikely(expr) __builtin_expect(!!(expr), 0)
#define likely(expr) __builtin_expect(!!(expr), 1)
if (likely(a > 1))
```

Hints for compiler: inlining

- `inline` doesn't force inlining, its just a hint for compiler
- `__attribute__((always_inline))` doesn't force it either, but in general compiler always follows this attribute, unless you take an address of function
- More inlining isn't automatically better:
  - No overhead for function call
  - More space for optimizations
  - But extensive inlining leads to a bigger codebase and thus less efficient usage of instructions cache

std

- Most of the topics are covered in std lecture
- Note, that some functions (mostly C functions) are either not thread-safe or introduce some internal locks
  - So never use `rand`, switch to generators from `<random>` instead
  - Some implementations of `std::to_string` may introduce partial serialization of calls
- I/O in std isn't good
- Especially avoid anything using locales internally, like `std::stringstream`

SPECTRAL:
TECHNOLOGIES

I/O

- Obviously we want to avoid any external I/O in low-latency programs. Sometimes we still need to log something though. We need to do 2 things fast:
  - converting output to text
  - writing this text
- std is bad at both
  - `ostream&` objects are not efficient at converting to text, since they use locales
  - `std::cout` needs a couple of special settings to start performing somewhat normally
  - `fstream` objects don't have this issue, but there is no way to do async output, which is the most important part

Logging

- Most fast logging libraries follow the same pattern:
  - Create a MPSC queue, which stores the messages to log
  - Create a separate thread (consumer), which does the actual writing. This thread should run on non-isolated cores with low priority
  - To log something, put the message into queue (no context switch, no blocking waiting for I/O)
- The same pattern can be applied to other background tasks as well

SPECTRAL:
TECHNOLOGIES

Template callbacks

```
using Data = std::variant<Type1, Type2, ...>;
Data ReadData() {
    ...
}

void ProcessData(Data *data) {
    std::visit([](auto&& arg) { // or use multiple std::get_if
        using T = std::decay_t<decltype(arg)>;
        if constexpr (std::is_same_v<T, Type1>) {
            HandleType1(arg);
        }
        ...
    },*data);
}

void f() {
    for(;;) {
        auto data = ReadData();
        ProcessData(&data);
        Update(data);
    }
}
```

Template callbacks

- `variant` internally keeps the index of the current type. `std::visit` builds a table `index -> function` and uses the index to call the corresponding for this type function
  - So its basically the same as using virtual functions
- The longer this pipeline this, more indirect function calls we get
- This issue arise from the fact, that we lose type information between each stage of this pipeline
- Ideally the type branching is only needed in `ReadData`

SPECTRAL:
TECHNOLOGIES

Template callbacks

```cpp
template<class Callback>
void ReadData(Callback callback) {
    ...
    callback(std::move(data));
}

template<class T, class Callback>
void ProcessData(T&& data, Callback callback) {
    if constexpr (std::is_same_v<T, Type1>) { // or use dispatching
        HandleType1(std::forward<T>(data), callback);
    }
}

void f() {
    for (;;) {
        ReadData([](auto&& arg) {
            using T = std::decay_t<decltype(arg)>;
            ProcessData(std::forward<T>(arg), Update);
        });
    }
}
```

SPECTRAL:
TECHNOLOGIES

Template callbacks

- This is also known as *continuation-passing style*
- We removed all of the dynamic dispatching in the pipeline and the code is still readable
- Templates can remove a lot of potential runtime branching in your code

Additional materials

- https://clang.llvm.org/docs/UsersManual.html#profile-guided-optimization
- https://llvm.org/docs/LinkTimeOptimization.html
- https://clang.llvm.org/docs/ThinLTO.html
- https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html (a lot of useful ones like `__builtin_popcount` and etc.)