Project Codebase

2024.04.18

SWPP Practice Session

Seunghyeon Nam

Summary

- We will distribute the following before the project
  - Per-team upstream repository with swpp-compiler skeleton code
  - swpp-interpreter repository
  - swpp Docker image
  - Alive2 repository with swpp customization

swpp-interpreter

- Executes the program written in swpp assembly
- Automatically calculate the total execution cost
- Shows the cost analysis for every function and instruction ran
- Crashes with error message upon encountering illegal program
  - Invalid syntax, illegal memory address, etc
- Note: the interpreter is written in Rust

swpp-interpreter

- You can test your optimizations with the interpreter
  - If the interpreter starts to yield wrong output,
    your optimization might be wrong
  - Comparing the execution cost before and after the optimization can show the effectiveness of it.

swpp Docker Image

- Most details are in the ‘Continuous-Integration’ slides
- One important update: Alive2 will not be included in the image
  - It is hard to update the CI image in case of urgent update
  - We’ll share the Alive2 repository instead
  - You can use `actions/checkout` to fetch the repository and build Alive2
  - You can use `actions/cache` to prevent frequent rebuilding

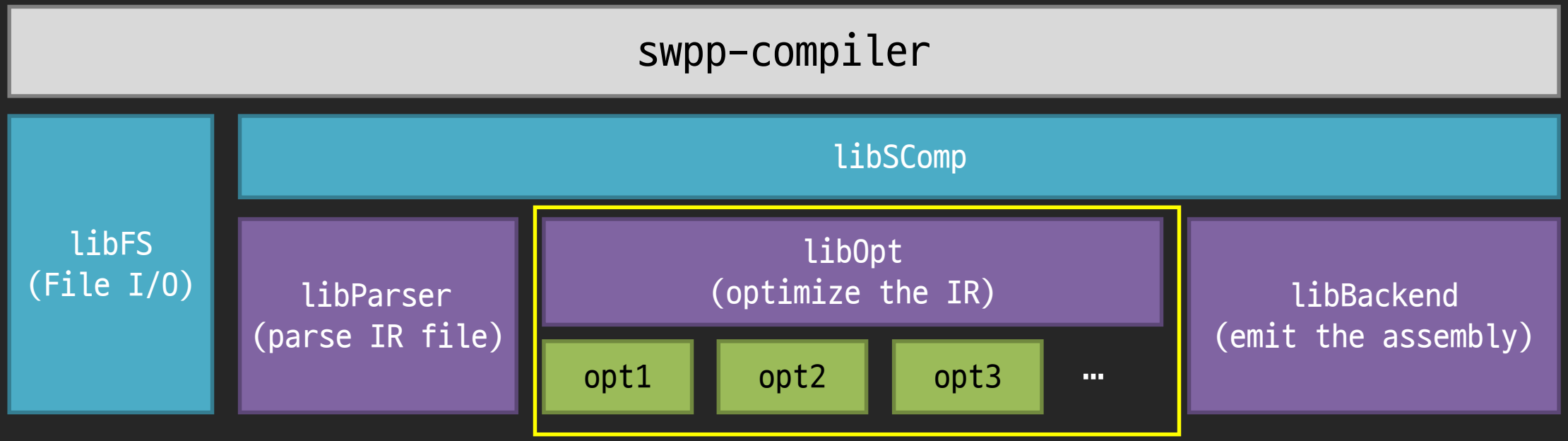
Alive2

- Verify the refinement between two LLVM IR programs
- Customizations are made to verify swpp-specific instructions
  - You cannot use the original Alive2 for the project
- Use small source and target program
  - It may fail to verify the optimization due to timeout
  - Or miss the incorrect optimization due to program complexity

swpp-compiler

- Your codebase for the team project
- Reads LLVM IR program from the file
- Applies your optimizations to the IR program
- Emits assembly from the optimized IR program
- Writes the emitted assembly back to file

swpp-compiler Overview



Team Project Scope

Codebase Characteristics

- Based on & makes extensive use of C++23 features
- Modularized structure
  - Don't worry about parser or backend during the project!
  - Your only concern should be IR-to-IR optimization
- CMake build system for easier configuration and testing
- In-source Doxygen documentation

Project Scope

- You're allowed to modify only a small fraction of the codebase
  - lib/opt.cpp
  - Files inside lib/opt directory
- Modifying the source outside the scope will be penalized
  - Ask the TAs if you really think you have no choice but to modify them

Project Scope

- No restrictions on new test files, scripts, or CI scripts
  - As long as they are not related to the compiler code, it's okay

swpp Intrinsics

- The swpp assembly language have some unique instructions
  - `incr, decr, vbcast, …`
- They don’t have the corresponding LLVM IR counterparts
- You have to use the intrinsics to emit those instructions
  - Compiler backend will convert these intrinsics into instructions

swpp Intrinsics

- Intrinsics can be called like ordinary LLVM IR functions
  - Ex) `%2 = call i64 @incr_i64(i64 %1)`
  - This will be converted to `r_ = incr r_ 64`
- Intrinsics must be ‘declared’ prior to its use
  - This is a restriction due to LLVM IR grammar
  - Note that you don’t have to ‘define’ these instructions

CMakeLists.txt

- Build script used by CMake
- Always update the CMakeLists when you add a new file
- There's a helper function inside for easier registration
  - Your passes will be built as an independent shared library
  - You can use the shared library to test with LLVM opt
  - See also: Assignment 4

Doxygen

- In-source documentation utility
- Converts the comments into documentation webpage
- Our codebase will include Doxygen documentation
- You're not required to documentize your passes with Doxygen
  - But it looks fancy ☺