Advanced Compiler Construction

Labs 2021

August Ernstsson (based on existing material by Erik Hansson)

LLVM Overview

- Low Level Virtual Machine – LLVM
- http://llvm.org
- Modern module-based compiler infrastructure
- Open Source
- Written in C++ (mainly) and also a lot of custom definition formats
- Started 2000 at University of Illinois at Urbana–Champaign by Chris Lattner and Vikram Adve.
- In 2005 Lattner got hired by Apple to work with LLVM.
- Clang: C/C++ language (and dialects) front-end for LLVM.
- LLVM and Clang are popular and successful, used extensively in industry and for academic research.

About Me

- Final-year PhD student at LiU.
- Course assistant in DF00100, helping with lab supervision.
- Research interests in high-level parallel programming, esp. with the skeleton programming approach.
    - Head developer/maintainer of the SkePU C++ template framework.
        - https://skepu.github.io
        - SkePU uses a custom source-to-source ”pre-compiler” based on the Clang library, using C++ AST traversal and analysis.
- I have experience with Clang, but not as much with the rest of LLVM, e.g. backend stuff. We will learn together!

About the Labs

- Work with the LLVM frameworks at different levels
  - IR analysis
  - Back-end code generation
- Relatively free-form lab format
  - Required programming: light
  - Encourages experimentation. Try your own ideas and extensions
  - Requires a lot of reading: LLVM documentation and source/sample code.
- Written reports each for part 1 and part 2
  - Document your work and your results. (More info later)

LLVM Getting Started

- In Linux, using git and CMake:
    - Clone LLVM source (Clang is now included in the repository) into **LLVM_DIR**
        - `git clone https://github.com/llvm/llvm-project.git`

    -
      ```
      cd LLVM_DIR
      git checkout release/10.x
      mkdir build
      cd build
      cmake -G "Unix Makefiles" -DLLVM_ENABLE_PROJECTS="clang" ../llvm
      make
      ```
    - (Optional) Add **`LLVM_DIR`**`/build/bin/` to your PATH
- LLVM may also work on Mac or Windows, see documentation.

Part 1 — LLVM IR

LLVM First Try

- Write a small C program
- Compile and run

    -
      ```
      LLVM_DIR/build/bin/clang -o test test.c
      ./test
      ```
    - Very similar to GCC
- LLVM IR

    -
      ```
      LLVM_DIR/build/bin/clang test.c -S -emit-llvm
      ```
        - This produces test.ll — investigate the output yourself!
    - Convert to ”bitcode”: **LLVM_DIR**/build/bin/`llvm-as test.ll`
        - Produces test.bc

LLVM Passes

- A pass can perform analysis or transformations on LLVM IR.
- Simple example from LLVM documentation
  - https://llvm.org/docs/WritingAnLLVMPass.html
    https://llvm.org/docs/WritingAnLLVMNewPMPass.html
- Lab ”exercise 0”:
  - For each function call in a program, print out its name.

Part 1, Exercise 1

- Write a simple pass that calculates how many calls there are to the printf() function.

- Output can e.g. be something like:

  -
    ```
    main:
      printf(): 2 calls
    foo:
      printf(): 23 calls
    ```

- It should handle simple loops with static iteration counts.

```
for (int i = 2; i < 4 ; i++)
{
  printf("Hello LLVM + CLANG!\n”);
}
```

Part 1, Exercise 2a

- Write a pass or set of passes that recognizes a vector init (initialization of an array with a constant value)
- Example:

```
for (int i = 0; i < 5; ++i)
{
  v[i] = 42;
}
```

- The pass should report
  - Matched computation, operand size, etc.
  - Alternatively replace matched loops by an equivalent function call.

Part 1, Exercise 2b

- **Optional** — with bonus points in the exam.
- Write a pass or set of passes that recognizes a dot product computation.
- See further details in the lab instructions.

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=0}^{n} a_i b_i$$

Part 2 — LLVM Back-end

Adding an instruction to LLVM

- In this assignment, you will add a theoretical instruction to the Sparc target.
- Create an add-instruction that takes three operands:
  - `addthree a,b,c,d`
- This instruction should match the computation d = a + b + c where
  - a,b,c are all integers located in registers,
  - a,b,c are all floats located in registers,
  - a,b are integers in registers, and c is an immediate value.

Part 2 — Hints

- You can compile your example program to sparc assembly like this:

-
  ```
  LLVM_DIR/build/bin/clang -O2 -S -emit-llvm foo.c
  LLVM_DIR/build/bin/llvm-as foo.ll
  LLVM_DIR/build/bin/llc --march=sparc --mcpu=generic --asm-verbose foo.bc
  ```
- Study foo.s to see the result.
- Declare your variables to be volatile:
  - `volatile int x;`

Written Reports — Requirements

- Your written report should contain the following:
  - Strategy and approach for solving the problems.
  - Results of your tests, with comments.
  - Implementation source code, commented where necessary.
  - Test programs, in C and LLVM assembly.
    - Also include invocation details, such as which passes were used and in which order.
  - Note which LLVM version was used and where you obtained it.
  - Some discussion around your results and your experiences.
- Send reports to august.ernstsson@liu.se
- Deadline information on course webpage.

General Hints

- LLVM and Clang are large projects with a lot of different contributors.
- Documentation quality varies a lot.
- Many tutorials available on LLVM.org and elsewhere.
    - May be out-of-date!