Project 1

Advanced Compilers 2024

TA: Juneyoung Park (`juneyoung.park@snu.ac.kr`)

May 9, 2024
Due: May 16, 2024

In this introductory project, we will skim through LLVM pass plugin. LLVM's middle-end IR-to-IR optimizer, `opt` manages passes, which are sort of function over IR. We will implement a simple instruction counting pass as a dynamic plugin. Since, integrating a new pass into `opt` needs rebuilding it, which may cost a long time, our pass will be implemented as out-of-tree, stand-alone pass plugin. Thankfully, `opt` can load pass plugin dynamically before executing it.

1 Background

1.1 LLVM IR

LLVM is renowned compiler infrastructure, which has its own SSA-formed IR language. LLVM language's main hierarchy consists of 'Module', 'Function', 'Basic Block', 'Instruction' (listed in wider to narrower order). Module is the largest unit of compilation pipeline, housing stores functions within itself using a doubly linked list. Similary, a function holds a list of blocks, and a block holds a list of instructions.

Among classes defined in LLVM API, one will likely be interested in '`Value`'. Various objects from `BasicBlock` to `Instruction` inherits the `Value`. Therefore, if you are interested in intra-procedure (function) level structures, you should keep in mind that many classes can be dynamically casted to `Value`. While casting those classes, LLVM encourages using its own implementation of `dynamic_cast`; `dyn_cast`. Also, note that the edges of SSA graph (or *Def-Use* chains) are embedded in `user` list of the `Value`. Please refer to the programmer's manual for more information.

1.2 Pass

A pass is a process which gets IR as input and also returns IR. A pass can be classified as a transform or analysis. If a pass does not modify the input IR, it is called as an analysis; otherwise, the pass is a transform. One can see various passes available in LLVM by '`opt --print-passes`'.

2 Instruction Counter

Workflow

1. While iterating over all basic blocks in the function, record hit count of each '`OpcodeName`'.
2. Print the statistics before exiting the pass.

2.1 Build

CMake is set for the project. One can generate build files for `ninja` build system by

```
cmake -S project1 -G Ninja -B build
```

Other build systems such as `Make` can be used by changing `-G` option. Also, build directory can be modified by configuring `-B` option.
After generation, one can finally build by calling

```
cmake --build build
```

2.2 Test

IR can be generated by `clang`, LLVM's C/C++ compiler, by

```
clang -O0 -S -emit-llvm -Xclang -disable-O0-optnone -fno-discard-value-
    names <source.c/cpp> -o <out.ll>
```

Built dynamic pass plugin is compiled as an `*.so` file (e.g., `build/lib/libInstructionCounter.so`). The plugin can be loaded and executed by

```
opt -S -load-pass-plugin='build/lib/libInstructionCounter.so' -passes='
    instruction-counter' <input.ll> -disable-output 2> test.out
```

Note that, `opt`'s standard output is used to print out result IR. Thus, if you want to print other text than IR, it is recommended to print to `stderr` by using LLVM's `raw_ostream`, `llvm::errs()`.

3 Objectives

1. Implement `TODO 1` at function '`run`' in '`InstructionCounter.cpp`'.
    - The pass should iterate over all the instructions and collect hit count of each opcode.

4 Submission

- Compress and submit your `InstructionCounter.cpp` as `PR1_<student_id>.zip` (e.g., `PR1_2024-12345.zip`) at eTL.

5 Example

```
Instruction hit counts
------------------------------
add : 2
call : 1
ret : 1
load : 6
br : 2
alloca : 4
store : 5
icmp : 1
------------------------------
```

Figure 1: Sample instruction counting pass' output