Assignment 6: LLVM IR Optimization (16%)

Due: April. 5, 2022 for Part 1

Note that this assignment allows resubmissions, but **no sample solutions** will be provided (for either part). Private test cases will be released after the submission deadline for part 1. The deadline for part 1 is April 5. The deadline for part 2 is the same as the resubmission deadline. We will use the resubmisison to collect the final programs that we use for testing speed. In this assignment, you will work on 2 parts for LLVM IR optimization in the MiniC Compiler:

- Memory to Register Promotion (`alloca2reg`) (11 marks out of 16)
- Open-ended IR optimization design (5 marks out of 16)

LLVM IR Optimization Workflow in MiniC (cont'd from Assignment 5 Workflow)

After the MiniC compiler IR Bitcode is generated, PassManager in LLVM will chain multiple optimizations and analysis passes to run together. Command `opt` is a LLVM tool that directly invoke a pass on a LLVM IR bitcode. You will work on the `src/Alloca2Reg.cpp` source file. `opt` will do a function pass based on `Alloca2Reg` library object on the IR bitcode.

alloca2reg Design

Please review **"Safely Promote Local Variable to Registers"** section from the lecture (under Files/Slides/Optimizations.pdf). It shows the detailed design steps. Lecture also provides a link How LLVM Optimizes a Function as your reference of the assignment.

alloca2reg Related Data Structures & LLVM API

Here is a good tutorial on Writing an LLVM Pass as a starting point. Table 1 shows the data structures in `src/Alloca2Reg.cpp`. You can add/modify/delete any structures in `src/Alloca2Reg.cpp`. We only provide some structures which may be helpful for you to implement alloca2reg. You are recommended to modify the structures for the open-ended design part.
Table 2 describes related LLVM APIs for alloca2reg.

Open-ended IR optimization design

In lecture, you were introduced to optimization methods such as machine dependent/independent optimizations. This section gives you some more resources for reference.

- LLVM Tutorial: Writing an Optimization for LLVM
- How LLVM Optimizes a Function as mentioned above
- Writing an LLVM Pass as mentioned above

Name	Description
struct Alloca2RegPass	Derived struct from llvm::FunctionPass.
Alloca2RegPass::ID = 0	Declares pass identifier used by LLVM to identify pass. This allows LLVM to avoid using expensive C++ runtime information. LLVM uses ID's address to identify a pass, so the initialization value is not important.
Alloca2RegPass::TargetAllocas & collectTargetAllocas()	Helps to collect all of the alloca instructions which can be removed. Please look at step 1 from the lecture.
Alloca2RegPass::Post & Pre	As mentioned in lecture, in step 2, they hold the representative value of the variable at the end of the basic block BB.
Alloca2RegPass::runOnFunction()	Overrides an abstract virtual method inherited from llvm::FunctionPass.
Other variables	To run an LLVM Pass automatically with clang. You don't need to worry about it. You could check out the link here for more details.

Table 1: Data Structure Description

Name under namespace llvm	Description
AllocaInst, StoreInst, LoadInst	alloca, store, load instructions
PointerType* AllocaInst::getType()	Overload to return most specific pointer type.
Type* PointerType::getElementType()	Return the type of the element which PointerType points to.
bool Type::isIntegerTy()	Return true if the type is integer type.
Value* StoreInst::, LoadInst::getPointerOperand()	Get the pointer operand.
Value* StoreInst::getValueOperand()	Get the value operand.
Instruction::eraseFromParent()	This method unlinks 'this' from the containing basic block and deletes it.
BasicBlock::begin(), end()	Instruction iterator methods
Function::begin(), end()	Basic Block iterator methods
PHINode member functions	You can search it for your implementation.

Table 2: LLVM API Description

You can create new source files for your optimizations OR implement it in `Alloca2Reg.cpp`. If creating new files, you may modify `src/CMakeLists.txt` as an additional library object. You should provide your command of generating IR bitcode when submitting.

Requirements

- The functionality of your output program should be correct.
- Your output program should remove all of the alloca instructions which can be removed.
- Your output program must reach a baseline speed under the -O0 flag. The baseline speed is the assignment 6 sample solution with only the alloca2reg optimization. As a reference, under these specs (*Memory: 32GB DDR4 & CPU: 3.5GHz 12-Core Intel Xeon W-2265*), the A6 sample solution with the -O0 flag running the `queen_time.c` test case with input=13 50 times consumes 63.82 seconds. clang-11 with -O3 flag consumes 22.77 seconds. The test script is provided as `asst6.py`.
- Your open-ended design speed/performance will be competing with the rest of the class. The marks for open-ended design will depend on your relative rank. Note that even the slowest implementation that is *functionally correct* will receive a passing grade for this assignment.

Compilation and Testing

Please make sure you are using `clang 11.0`. After compiling your MiniC compiler, you can run:

```
# Under build/ directory
# Generate IR Bitcode
src/minicc  <YourInput.c> -o output.bc
# Generate executable file
clang output.bc minicio/libminicio.a -o out
# Run Alloca2Reg pass
opt -O0 -load src/liballoca2reg.so --alloca2reg output.bc -o output_opt.bc
# Run the executable file
./output_opt
# Generate IR assembly code (Important for debugging!)
llvm-dis output.bc -o output.ll
llvm-dis output_opt.bc -o output_opt.ll
```

Public Autotester

We have provided a public autotester `a6tester.zip` on Quercus. After compiling your code, change `asst6.py` to your minicc executable file path. Then simply run:

```
    $./asst6.py
OR in verbose mode
    $./asst6.py -v
```

The tester has several MiniC source files. Your compiler will compile the files and the generated executable files will run without any errors in the tester. Then, compared to your unoptimized version, the tester simply checks whether the number of `alloca` instructions were reduced. Finally, run time test for your solution and -O3 clang solution as your reference.

Deliverables

In the project folder, compress the whole project content as a `zip` file. This zip file must match the structure of the skeleton code, and it must compile in our A1 environment, or you may receive a 0 on the automarker. On Markus, please submit:

- `code.zip`. We will directly unzip your submission, build it from source and run it against public and private tests. Before submitting, please change the format for including llvm headers to `#include "llvm/***"` instead of some personal formats such as `#include "llvm-11.0/***"` or `#include "/usr/local/llvm/***"`.
- `explanation.txt`: **Put your command for generating output_opt.bc in the first line, i.e., "opt -O0 -load ...." . You must use -O0 or don't use any flag in the command. Your command is used for testing your output program speed. Do not add anything ahead of your command such as comments! Otherwise, you may get zero since autotester will parse your first line as command run!** Then put your explanation of optimization methods in a newline.
- `sample.c`: A sample program which you think performs well under your optimizations. Its complexity and its performance improvement will be considered for grading.