Assignment 3: Code Optimization and Profiling

Due Date: Apr. 5th, Total Marks: 100 pts

CSCD70H3 S (Winter) Compiler Optimization
Department of Computer Science, UTSC

ABSTRACT

In this assignment, you will write passes to improve code by LICM. To convince yourself of the benefits of your code transformations, you will measure the resulting program speedups.

1 POLICY

1.1 Collaboration

You will work in groups of two people to solve the problems for this assignment (you can also choose to work individually). **Please turn in a single writeup per group, indicating the names and UTORid of all group members.**

1.2 Submission

Please include the following items in an archive labeled with the UTORid of all group members (`a3-utorid1(-utorid2).tar.gz`), and submit the resulting file via *Markus* (`https://markus.utsc.utoronto.ca/cscd70w18/`). Please make sure that when this archive is extracted, the files appear as follows:

```
./a3-utorid1(-utorid2)/writeup.pdf

./a3-utorid1(-utorid2)/licm/licm.cpp
./a3-utorid1(-utorid2)/licm/Makefile
./a3-utorid1(-utorid2)/licm/tests/
```

- A report named **writeup.pdf** that briefly describes the implementation of the framework and both passes, and has answers to the theoretical questions in this assignment.
- Well-commented source code for your optimization pass **licm**, and associated **`Makefile`** (please write your `Makefile` in such a way that the pass can be built (and run) using the command `make all`).
- A subfolder named **`tests`** that contains all the microbenchmarks that were used for verification of your code.

2 PROBLEM STATEMENT

2.1 Loop Invariant Code Motion [60 pts]

In this pass, you will decrease the number of dynamic instructions executed during a loop by identifying and hoisting out those that are **loop-invariant**, as was discussed in class. Please call your pass **licm**.

In addition to the usual preprocessing *mem2reg*, you should also optimize your code using the LLVM built-in pass ***loop-simplify*** to insert loop preheaders where appropriate. If this built-in pass is unable to insert a preheader (i.e. **`loop->getLoopPreheader() == nullptr`**), you can ignore the loop.

It is recommended to derive from LLVM's **`LoopPass`**. You are encouraged to use LLVM's built-in pass **`DominatorTreeWrapperPass`** and **`LoopInfoWrapperPass`** (please refer to the second tutorial on how to achieve this using the LLVM's optimization manager). **You are not allowed to use methods related to loop-invariance**, which includes but not limited to **`isLoopInvariant`**, **`hasLoopInvariantOperands`**, **`makeLoopInvariant`**. **You can, however, learn from LLVM's way of doing this and write your own equivalent implementation.**

For each loop, compute the set of loop-invariant instructions. You may ignore child nested loops that you have already processed, but **you should ensure that deeply-nested loop-invariant computations can still bubble all the way out**. When checking for loop-invariance, you should also include the following **additional conditions** in Listing 1 for determining whether an instruction is invariant. Hoist to the preheader all loop-invariant instructions that are candidates for code motion, ensuring that dependencies are preserved.

```
bool isInvariant(Instruction * I)
{
    bool is_invariant = // your
    implementation goes here

    return isSafeToSpeculativelyExecute(I)
        && !I->mayReadFromMemory()
        && !isa < LandingPadInst > (I)
        && is_variant;
}
```

Listing 1: `isInvariant` Code Snippet

In your documentation, please make sure to **describe at least the following three points**: (1) how you check for loop invariance, and also, why you think we need the additional conditions in Listing 1 (especially the first two) (2) how you hoist the code that is loop-invariant to the loop preheader (3) how you handle cases where there are nested **`for`**-loops .

2.2 LLVM Profiling [20 pts]

Programs can be profiled in multiple ways. You could simply time your program over some number of iterations, but your results would be highly dependent on your particular machine's hardware and software configuration; however, this requires no changes to be made to the program under inspection.

Another way to estimate the performance of a program is to simply measure how many LLVM instructions are dynamically executed when it runs. To do this, you can use *lli*, the LLVM interpreter. Ordinarily the interpreter will try to Just-In-Time (JIT) compile the bytecode passed to it, but you can force it to take the slow path (while counting instructions) by using the command:

```
$ lli -stats -force-interpreter my_test.bc
```

You should always get the same instruction count every time you run *lli*. This approach works best with test programs that have a **`main()`** function. This is, of course, not a very good machine model - for example, all instructions are assigned the same cost (even pseudo-instructions, like **`getelementptr`**) and there is no notion of memory latency. As a first pass, though, it provides a nice way to measure the effectiveness of your passes. In your writeup, please **discuss the changes in dynamic instruction count on the transformed bytecode after running your pass on each microbenchmark.**

3 THEORETICAL QUESTIONS

3.1 Register Allocation [20 pts]

Suppose that you have a processor with four registers. Consider the following code, where only definitions and uses of interest are shown. Perform the **register allocation** algorithm described in class, showing the following steps for full marks: (1) live variables (2) reaching definitions (3) live ranges (4) interference graph (5) final colored graph .

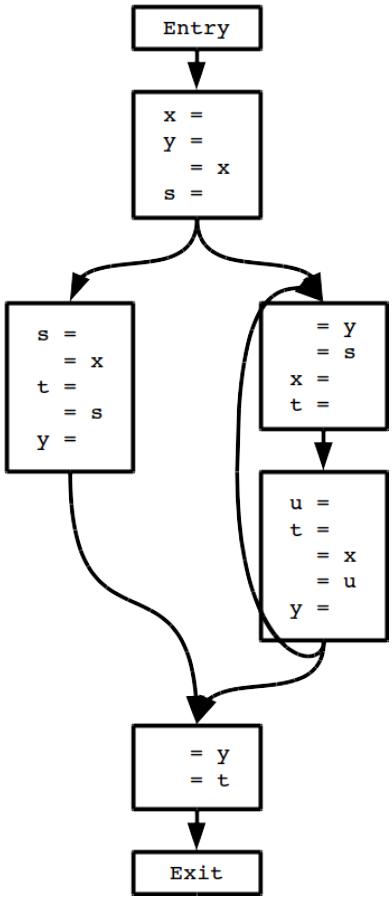

Figure 1: Code for Register Allocation Analysis