CS 745, Spring 2012
Homework Assignment 3: Code Transformations

Assigned: Thursday, February 16
Due: Thursday, March 8, 9:00AM

Introduction

In this assignment, you and your partner will get the chance to use your compiler analysis writing skills to improve code by eliminating redundant computations. Furthermore, to convince yourself of the benefits of your code transformations, you will measure the resulting program speedups.

Specifically, you will implement Dead Code Elimination (*DCE*) in your iterative dataflow analysis framework (from the previous assignment). You will then use your DCE pass to eliminate the redundant computations in unoptimized LLVM bytecodes. Fortunately, unlike registers, memory objects are not represented in SSA form in LLVM, and since memory references are more prevelant in unoptimized bytecodes, dealing with ϕ functions should be less of an issue compared to the previous assignment.

An interesting twist to your DCE pass is that it will use a more sophisticated Liveness analysis compared to what we discussed in class. This variant of Liveness—which we will call *Faint Variable Analysis*—will be described in more detail below.

Finally, you will measure the impact of your optimizations on the execution times of programs. Further details that are relevant for completing this assignment are provided below.

Policy

You will work in groups of two people to solve the problems for this assignment. Turn in a single writeup per group, indicating all group members.

Logistics

Any clarifications and revisions to the assignment will be posted on the "assignments" page on the class web page.

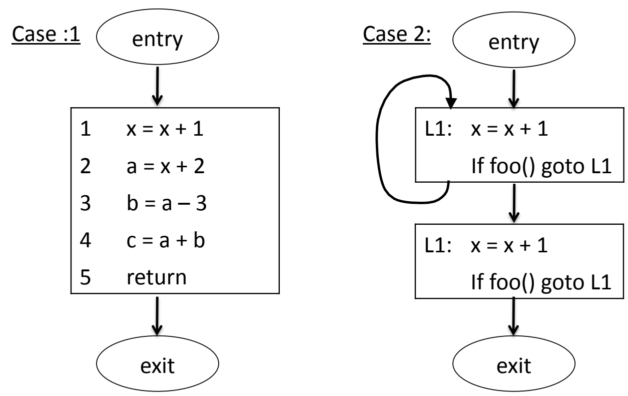
In the following, *HOMEDIR* refers to the directory:

`/afs/cs.cmu.edu/academic/class/15745-s12/public`

and *ASSTDIR* refers to the subdirectory *HOMEDIR*`/asst/asst3`.

1 Dead Code Elimination

The idea behind Dead Code Elimination (*DCE*) is that an assignment of the form "`x = t`" can be eliminated if its LHS variable `x` is not live (i.e dead) at the program point P immediately following the assignment. One of the limitations of DCE is that it cannot directly eliminate the assignment "`x = x + 1`" in the two examples shown below:



In the first case, `x` is not dead after the "`x = x + 1`" assignment (instruction 1) because it is used in instruction 2. Instruction 2 is also not dead because its LHS variable (`a`) is used in instructions 3 and 4. However, instruction 4 is in fact dead. If we applied DCE repeatedly to this code, we could eventually eliminate instruction 1. However, it would be more desirable to eliminate "`x = x + 1`" in a single data flow pass.

In the second case, the LHS of "`x = x + 1`" is not dead because it is used by its own RHS due to the cycle in the flow graph. However, since the ultimate value of `x` is never used, this instruction could in fact be safely eliminated from the loop body.

We say that the LHS variable `x` in an assignment "`x = t`" is *faint* if along every path following the assignment, `x` is either dead or is only used by an instruction whose LHS variable is also faint. Your mission in this assignment is to write a new data flow analysis called *Faint Variable Analysis* (FVA) that will directly determine that the LHS variable of "`x = x + 1`" is "*faint*" in both of these cases.

Some issues to consider when designing your FVA algorithm:

1. What is the direction of the data flow analysis ?
2. What is the meet operator ?
3. What are the lattice elements ?
4. What are the values of top and bottom ?
5. How do you initialize the iterative algorithm ?
6. The transfer function (hint: this must be done at the instruction level, not the basic block level).

Now, write a DCE pass that uses FVA to eliminate instructions that have either faint or dead LHS variables.

1.1 Measure Impact of Optimizations

Finally, you should evaluate the effectiveness of your optimizations in improving the execution time of programs. At the minimum you should write a few synthetic benchmarks, and measure how much your optimizations can speed them up. It might also be worthwhile to compare your DCE pass against the LLVM version.

For your convenience, code that is similar to Case 1 above will be distributed with this assignment. Consult the `Makefile` for steps on generating and running LLVM executables.

Note: Make sure that your pass is called `dce-pass`, and that it is the only one requried to execute all of your optimizations.

2 LICM: Loop Invariant Code Motion

The basic idea behind Loop Invariant Code Motion (LICM) is that certain expressions in the loop stay the same (or invariant) during the loop execution, and, hence, it is possible in some cases to avoid executing these code multiple times. Lecture 8 describe the Code Motion algorithm to achieve this. The key steps consists of:

1. Compute Reaching Definitions (using your dataflow framework from Assignment 2)
2. Compute Loop Invariant Computation
3. Find the exits from the loop (i.e., nodes with successors outside the loop)
4. Candidate statement for code motion:
   (a) loop invariant
   (b) in blocks that dominate all the exits of the loop
   (c) assign to variable not assigned to elsewhere in the loop
   (d) in blocks that dominate all blocks in the loop that use the variable assigned
5. Perform a depth-first search of the blocks. Move candidate to preheader if all the invariant operations it depends upon have been moved

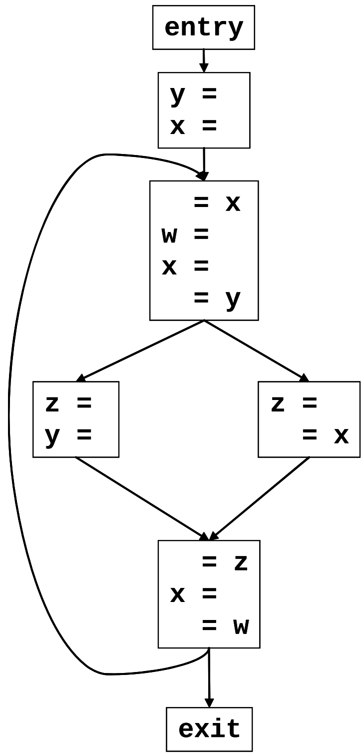
Note: You are allowed to use LLVM loop structures to find appropriate loops (the pass name is *-loops*, corresponding classes are defined in *LoopInfo.h*) or you are free to implement loop detection yourself.

Make sure that your pass is called `licm-pass`, and that it is the only one requried to execute all of your optimizations.

3 Questions

3.1 Register Allocation

Consider the following code, where only uses and defs of interest are shown:



1. Rewrite the code using only four registers r1, r2, r3 and r 4 instead of variables w, x, y, and z. Draw the interference graph. If you must spill, clearly identify what you spilled and why that was chosen.

2. Could you allocate all variables using only three registers? If not, what would you spill? Why?

3.2 Instruction Scheduling

Imagine that you had a processor with two independent arithmetic units and one load/store unit. Adds and subtracts take 1 cycle, multiplies take 3 cycles, divides take 5 cycles, loads take 2 cycles, and stores take 1 cycle to initiate. Assume that the processor can only issue two instructions per cycle, and that each functional unit can accept one instruction per cycle.

1. Show how the forward list scheduling algorithm described in class would execute the code in Listing 1 on the processor described above. Assume the priority mechanism described in class, and that ties will be broken by selecting the instruction that appeared first.

   You should turn in a table, where each row represents a cycle, containing the number of that cycle (starting with 0), what instructions are ready to execute, and what instruction will be issued to each functional unit.

2. Show how an optimal scheduling algorithm would schedule the same instructions, where in this case optimal means that the processor will finish executing the instructions in as few cycles as possible. Do not worry about how many cycles stores take to actually write to memory - assume that they are complete once the store instruction has been issued.

   You should turn in a table, where each row represents a cycle, containing the number of that cycle (starting with 0), what instructions are ready to execute, and what instruction will be issued to each functional unit.

```
a = load A
b = load B
c = a / b
d = a+b
e = load E
f = a+a
store f,F
g = d+e
h = load H
i = c*e
j = d+h
store h,H
store d,D
store j,J
store g,G
```

Listing 1: Code for instruction scheduling

4 Hand In

Electronic submission:

- The source code for your code transformation passes, the associated `Makefiles`, and a `README` describing how to build and run them. Do this by creating a tar file with the last name of at least one of your group members in the filename, and copying this tar file into the directory

  `ASSTDIR/handin`

Include as comments near the beginning of your source files the identities of all members of your group. Also remember to do a good job of commenting your code.

- Results showing the impact of your DCE and LICM passes on the execution time of LLVM executables, including the source code of benchmarks.

Hard-copy submission:

1. A report that briefly describes the design and implementation of the code transformations, as well as your performance evaluation strategy.
2. A listing of your source code.
3. A set of tests/benchmarks (including the source code) that you used to test your passes.
4. A hard copy of the solutions for questions in Section 3.