15-745, Spring 2013
Homework Assignment 3

Assigned: Thursday, February 14
Due: Thursday, March 7, 9:00AM

Introduction

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

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

All clarifications (if any) to this assignment will be posted on the class discussion board on Piazza. Any revisions will be uploaded to the "assignments" page on the class web page.

0.1 Changelog

- (3/1) To get around some issues with sequencing transform passes in `opt`, we will ensure that LoopSimplify runs before your LICM pass by running it with:

  ```
  opt -load ./ClassicalDataflow/loop-invariant-code-motion.so -loop-simplify -cd-licm
  ```

- (2/28) You are not required to use `LoopPass` for this assignment, but it is a reasonable idea to do so.

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

```
/afs/cs.cmu.edu/academic/class/15745-s13/public
```

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

1 Dead Code Elimination

In this pass, you will improve the performance of LLVM functions by removing unnecessary ("dead") instructions and preserving live ones. An `Instruction* I` is live if:

```
isa<TerminatorInst>(I) ||
isa<DbgInfoIntrinsic>(I) ||
isa<LandingPadInst>(I) ||
I->mayHaveSideEffects()
```

or if `I` is used by any `Instruction` that is also live. Compute the set of instructions to remove or preserve, then use it to eliminate instructions appropriately. Use your dataflow analysis framework to process sets of instructions simultaneously.

(Note: you will need to include `llvm/IntrinsicInst.h` to get all of the definitions you need for the above expression.)

Your pass must be named `cd-dce` and we must be able to run it with `opt -load ./ClassicalDataflow/dead-code-elimination.so -cd-dce`

Write *at least three* microbenchmarks and *include in your writeup* a listing of the original C source, the LLVM bitcode (after being transformed by `mem2reg`) with a dynamic instruction count (described in the next subsection), and the transformed LLVM bitcode (put through your pass after being transformed by `mem2reg`) with a dynamic instruction count.

1.1 Profiling with LLVM

LLVM supports profiling in various 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. You can also use LLVM's intrusive profiling support; for example, to collect edge profiling information:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:‘llvm-config --libdir‘
clang -emit-llvm -O3 -o foo.o -c foo.c
opt -insert-edge-profiling -o foo.inst.bc foo.o
clang -L‘llvm-config --libdir‘ -lprofile_rt foo.inst.bc -o foo.native
./foo.native
llvm-prof -stats foo.inst.bc llvmprof.out
```

`opt` inserts calls to functions defined in the `profile_rt` shared library. When the instrumented program is run, it adds samples to `llvmprof.out`. This information can be printed using the `llvm-prof` tool.

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 JIT compile the bitcode passed to it, but you can force it to take the slow path (while counting instructions):

```
clang -emit-llvm -O3 -o foo.o -c foo.c
lli -stats -force-interpreter foo.o
```

You should always get the same instruction count every time you run `lli`. 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 DCE and LICM passes.

2 LICM: Loop Invariant Code Motion

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.

For this problem, you may derive from LLVM's `LoopPass` class. You may also require and use loop information from the `LoopInfo` pass except for methods related to loop-invariance. Finally, you should rely on the `LoopSimplify` pass to insert loop preheaders where appropriate. (We will run your pass in such a way that `LoopSimplify` executes before it.) If `LoopSimplify` is unable to insert a preheader (eg, `((Loop*)foo)->getLoopPreheader() == NULL`), you may ignore the loop.

You must write your own code to calculate dominance information for this problem. For each loop you process, print out the names and immediate dominators of the blocks belonging to the loop (to standard error; names should come from `BasicBlock::getName`).

```
for.cond idom entry
for.body idom for.cond
for.inc idom for.body
```

You may find your dataflow analysis framework to be useful for this subproblem, but you are not required to use it.

Next, 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. Follow the guidelines from the class notes, with these added stipulations for determining whether LLVM `Instruction* I` is invariant:

```
  isSafeToSpeculativelyExecute(I) &&
  !I->mayReadFromMemory() &&
  !isa<LandingPadInst>(I)
```

In your writeup, describe why the first two checks are necessary. `isSafeToSpeculativelyExecute` is defined in `llvm/Analysis/ValueTracking.h`.

Finally, hoist to the preheader all loop-invariant instructions that are candidates for code motion. Be sure to preserve dependencies.

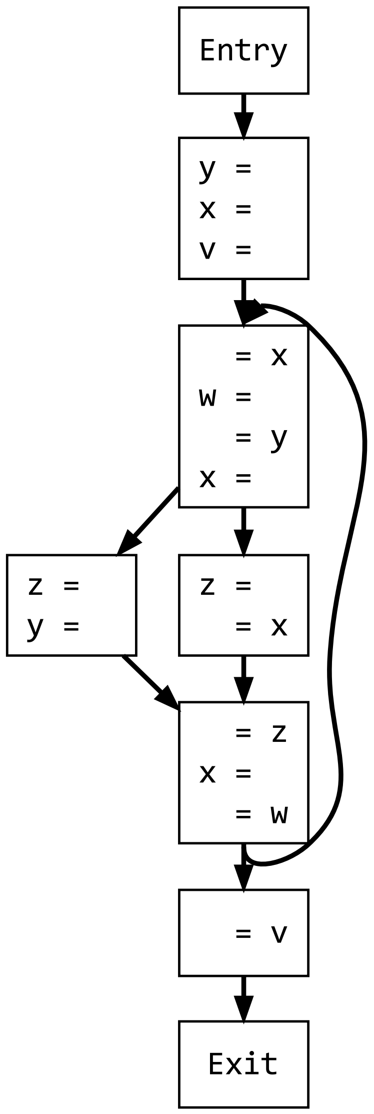
Your pass must be named `cd-licm` and we must be able to run it with `opt -load ./ClassicalDataflow/loop-invariant-code-motion.so -loop-simplify -cd-licm`

Write *at least three* microbenchmarks and *include in your writeup* a listing of the original C source, the LLVM bitcode (after being transformed by `mem2reg`) with a dynamic instruction count, and the transformed LLVM bitcode (put through your pass after being transformed by `mem2reg`) with a dynamic instruction count.

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 `r4` instead of variables `v`, `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. All memory addresses are distinct.

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 program order.

2. Repeat the previous exercise, this time using the reverse list scheduling algorithm described in class.

For each task, you should turn in a table with one cycle per row. For each cycle, give the number of the cycle (starting from 0), which instructions are ready to execute, and which instructions 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 d, D
g = d + e
h = load H
i = c * e
j = i + h
store h, H
store f, F
store j, J
store g, G
```

Listing 1: Code for instruction scheduling

4 Hand In

Hard-copy submission:

1. A report that briefly describes the design and implementation of the code transformations, as well as reasoning behind the design of your benchmarks.
2. A listing of your source code.
3. A set of tests/benchmarks (including the source code) that you used to test your passes, with performance information as described in parts 1 and 2.
4. Your answers to the questions in parts 2 and 3.

Electronic submission:

- A PDF of your writeup report and answers to the questions, named `writeup.pdf`.
- The source code for your code transformation passes, the associated `Makefile`s, your benchmarks and test cases, and a `README` describing how to build and run them (especially if you for some reason diverge significantly from what we have asked you to do; note, however, that you are expected to produce the files and use the pass names we require).

  Place all of these files in a directory with the same name as the Andrew ID of one of your group members. Archive this directory and name the file with the same Andrew ID (`bovik.tar.gz`):

```
tar czvf bovik.tar.gz bovik
```

When the file is extracted with `tar xf`, we expect to see these required files in these locations:

```
./bovik/README
./bovik/ClassicalDataflow/Makefile
./bovik/writeup.pdf
```

The `Makefile` must produce `./bovik/ClassicalDataflow/dead-code-elimination.so` and `./bovik/ClassicalDataflow/loop-invariant-code-motion.so`.

It is fine if there are other files included; please also include any additional tests you used for verification. Note that the sample `Makefile` assumes that your dataflow framework is still in `./bovik/ClassicalDataflow/dataflow.h` and `.cpp`.

If you need to update your submission but are not able to overwrite the old file, you may append a version number (separated by a dash) to the Andrew ID in the filename: `bovik-2.tar.gz` We will grade the most recent submission.

Copy the tar.gz file into the directory

```
/afs/cs.cmu.edu/academic/class/15745-s13/public/asst/asst3/handin
```

Include as comments near the beginning of your source files the identities of all members of your group. Please don't forget to comment your code.