15-745, Spring 2015
Homework Assignment 3

Assigned: Thursday, February 12
Due: Thursday, March 5, 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. Turn in all of your work both electronically and on paper.

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-s15/public`

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

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 machines 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 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 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.

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 LLVMs `LoopPass` class. You may also require and use loop information from the `LoopInfo` pass except for methods related to loop-invariance (`isLoopInvariant`, `hasLoopInvariantOperands`, `makeLoopInvariant`, etc.). 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 example:

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

We will ensure that `LoopSimplify` runs before your LICM pass by running it with:

```
opt -load ./ClassicalDataflow/LoopInvariantCodeMotion.so -loop-simplify -licm
```

You are not required to use `LoopPass` for this assignment, but it is a reasonable idea to do so.

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, please describe why the first two checks are necessary.

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 `licm` and we must be able to run it with
`opt -load ./ClassicalDataflow/LoopInvariantCodeMotion.so -loop-simplify -licm`

Write at least three microbenchmarks and include in your submission 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.

2.1 Dead Code Elimination

In this pass, you will improve the output of your LICM pass by removing unused (“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 files to get all of the definitions you need for the above expression.) Your pass must be named `dce` and we must be able to run it with:

```
opt -load ./ClassicalDataflow/DeadCodeElimination.so -dce
```

Write at least two microbenchmarks and include in your submission 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:

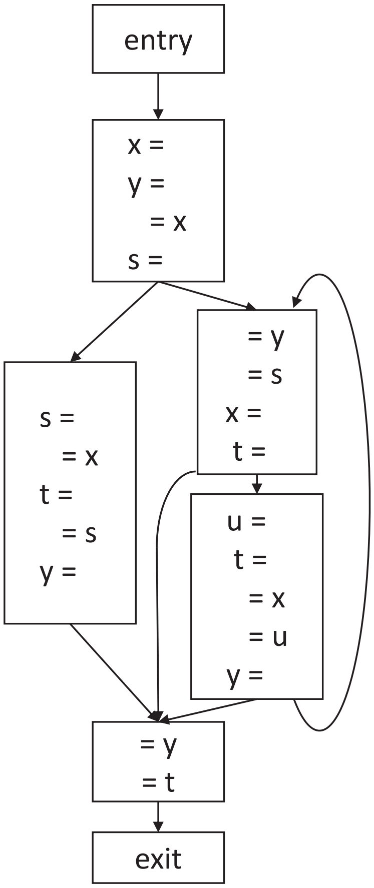

Figure 1: Code for Register Allocation Question

Using the algorithms discussed in class (see class notes from Lecture 15), show how to go about allocating registers using that algorithm. Show what data you are keeping track of after every step. Assume you have four physical registers to work with.

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 4 cycles, divides take 7 cycles, loads take 3 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 in the regular program order[1].

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

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

Listing 1: Code for instruction scheduling

[1] For reverse list scheduling, this means that if A appears before B in the original program, then you should select A to *finish* before B, even though this may make B appear before A when looking at your results in the forward direction.

4 Hand In

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.~~ You do not need to print out your source code or your test cases. Your electronic copy will be the one that is graded.
3. A set of tests/benchmarks (including the source code) that you used to test your passes, as well as the reasoning behind the design of your benchmarks.
4. A hard copy of the solutions for questions.

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 `Makefiles`, and a `README` describing how to build and run them (however, your directions should not diverge drastically from what we have asked you to do). 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 with the same Andrew ID. For example, if your Andrew ID is `bovik`, then you could archive it with `tar czvf bovik.tar.gz bovik`.

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

  ```
  ./bovik/README
  ./bovik/LICM/Makefile
  ./bovik/writeup.pdf
  ./bovik/tests/[your tests here]
  ```

  It is fine if there are other files included. Both the source code and the bitcode files for your tests and benchmarks should be put in the `tests` folder. The instruction count information should be included as part of your writeup PDF.

  Note: You will not be able to modify or replace your submission once you put it in the `handin` folder. If you need to update your submission, please add a new tgz file with a version number at the end, such as bovik-v2.tar.gz.

  ```
  ASSTDIR/handin
  ```

  Include as comments near the beginning of your files the identities of all members of your group. Please dont forget to comment your code.