Abstract
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.

1 Introduction

1.1 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.

1.2 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 class web page.

1.3 Submission
Please include the following items in an archive labeled with the Andrew ID of one member in your group (e.g., bovik.tar.gz), and submit the resulting file to Blackboard. Ensure that when this archive is extracted, the files appear as follows:

./bovik/README
./bovik/LICM/Dataflow.cpp
./bovik/LICM/Dataflow.h
./bovik/LICM/DeadCodeElimination.cpp
./bovik/LICM/LICM.cpp
./bovik/LICM/Makefile
./bovik/writeup.pdf
./bovik/tests/

- A report that briefly describes the implementations of both passes, and answers the homework questions, named writeup.pdf, containing the Andrew ID's of all members in your group.
- Well-commented source code for your passes (DeadCodeElimination and LIM), and associated Makefiles.
- A README file describing how to build and run your passes.
- Any tests used for verification of your code.
2 Profiling with LLVM

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 JIT compile the bitcode passed to it, but you can force it to take the slow path (while counting instructions): lli -stats -force-interpreter foo.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.
3 Analysis Passes

3.1 Dominators (Purple Dragon Book 9.6.1)

You will first need to implement your own pass to calculate dominance information over loops. Please call your pass dominators. It is recommended to derive from LLVM’s LoopPass class. For each loop you process, print out the names and immediate dominators of the blocks belonging to the loop (to standard output; names should come from BasicBlock::getName()). You should use your dataflow analysis framework to implement this pass. For an example, refer to Section 5.1 in the Appendix.

3.2 Loop Invariant Code Motion (Purple Dragon Book 9.5.1)

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 discussed in the lectures from class. In addition to hoisting out candidate instructions that dominate all exits of the loop, you will need to implement the landing pad transformation discussed in the lecture to handle loops with a zero iteration path through the loop. This will allow candidate statements that do not dominate all exits to be hoisted into the new preheader. Please call your pass loop-invariant-code-motion.

It is recommended to derive from LLVM’s LoopPass class. You should use dominator information provided by your pass, and loop information provided by the built-in LoopInfo pass, except for methods related to loop-invariance (isLoopInvariant(), hasLoopInvariantOperands(), makeLoopInvariant(), etc.). Refer to the LLVM documentation for specifying interactions between passes.

You may assume that the input received by your pass has already been transformed by the loop-simplify pass to insert loop preheaders where appropriate. If this built-in pass is unable to insert a preheader (e.g. ((Loop*) foo)->getLoopPreheader() == NULL), you may ignore the loop.

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. Use the guidelines from the class notes, with the following additional conditions (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.

```c
bool isInvariant(Instruction *I) {
    return isSafeToSpeculativelyExecute(I) && !I->mayReadFromMemory() &&!
            isa< LandingPadInst >(I);
}
```

Listing 1: Code for Pass

Include at least three microbenchmarks that you wrote in your submission. In your writeup, please describe why the first two of these checks above are necessary, and discuss the changes in dynamic instruction count on the transformed bitcode after running your pass on each microbenchmark.

3.3 Dead Code Elimination (Purple Dragon Book 9.1.6)

In this pass, you will improve the output of your LICM pass by removing unused (dead) instructions and preserving live ones. An instruction is live if it satisfies the following conditions (Listing 2), or
if it used by any live instruction. Compute the set of instructions to remove or preserve, then use this to eliminate instructions appropriately. Use your dataflow analysis framework to process sets of instructions simultaneously. Please call your pass **dead-code-elimination**.

```c
bool isLive(Instruction *I) {
    return isa<TerminatorInst>(I) || isa<DbgInfoIntrinsic>(I) || isa<LandingPadInst>(I) || I->mayHaveSideEffects();
}
```

Listing 2: Code for Pass 2

Include at least two microbenchmarks that you wrote in your submission. In your writeup, discuss the changes in dynamic instruction count on the transformed bitcode after running your pass on each microbenchmark.
4 Homework Questions

4.1 Instruction Scheduling (Purple Dragon Book 10.2 - 10.3)

Suppose that you have a processor with two independent arithmetic units and one load/store unit. Adds and subtracts take 2 cycle, multiplies take 3 cycles, divides take 4 cycles, loads take 1 cycles, and stores take 2 cycle. Assume that the processor can only issue two instructions per cycle, and that each functional unit can accept one instruction per cycle.

For each of the following problems, submit a priority-labeled data precedence graph, and fill in the scheduling table as shown below.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>ALU 0</th>
<th>ALU 1</th>
<th>LD/STR1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. Show how the forward list scheduling algorithm would execute the code in Listing 3 on the described processor. Assume the priority mechanism described in class and that ties will be broken by selecting the instruction that appeared earliest in the regular program order.

2. Show how the backwards list scheduling algorithm would execute the code in Listing 3 on the described processor. Assume the priority mechanism described in class and that ties will be broken by selecting the instruction that appeared latest in the regular program order.

```
1 a ← LOAD A
2 b ← LOAD B
3 c = MUL a, b
4 d ← LOAD D
5 e = SUB d, c
6 f = DIV b, d
7 STORE e → E
8 g = MUL e, f
9 h = ADD c, c
10 STORE g → G
11 STORE h → H
```

Listing 3: Code for Question 3
4.2 Register Allocation (Purple Dragon Book 8.8)

Suppose that you have a processor with four physical registers. Consider the following code, where only definitions and uses of interest are shown. Perform the register allocation algorithm described in class, showing each step (live variables, reaching definitions, live ranges, interference graph, final colored graph).

```
entry
x = y = = x
s =

s = = x
t = = s
y =

u = t = = x
= u
y =

= y
= t

exit
```

Figure 1: Code for Question 4.2
5 Appendix

5.1 Dominators: Example Input

```c
int loop() {
    for (int i = 0; i < 100; i++) {
        // do nothing
    }
    return 0;
}
```

Listing 4: Example input code.

```asm
define i32 @loop() #0 {
    entry:
        br label %for.cond
    for.cond:
        %i.0 = phi i32 [ 0, %entry ], [ %inc, %for.inc ]
        %cmp = icmp slt i32 %i.0, 100
        br i1 %cmp, label %for.body, label %for.end
    for.body:
        br label %for.inc
    for.inc:
        %inc = add nsw i32 %i.0, 1
        br label %for.cond
    for.end:
        ret i32 0
}
```

Listing 5: Example simplified input bitcode.

5.2 Dominators: Example Output

entry dom for.cond
entry dom for.body
for.cond dom for.body
entry dom for.inc
for.cond dom for.inc
for.body dom for.inc
entry dom for.end
for.cond dom for.end

Figure 2: Example analysis output.