Assignment #3

15-745 Spring 2019

Assigned: Wednesday, February 20
Homework problems due: Friday, March 8, 11:59PM
DCE and LICM implementation due: Monday, March 18, 11:59PM

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.

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.

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, Lecture 8)

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 `FunctionPass` 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 Dead Code Elimination (Purple Dragon Book 9.1.6)

In this pass, you will remove unused (*dead*) instructions and preserve live ones, using the faint analysis discussed previously. An instruction is live if it satisfies the following conditions (Listing 1), or if it is 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. Please call your pass **dead-code-elimination**.

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

Listing 1: Code for Pass 1.

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.

3.3 Loop Invariant Code Motion (Purple Dragon Book 9.5.1, Lecture 9)

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**.

For more details on the landing pad transformation, please see Section 4.2 in "Global Value Numbers and Redundant Computations" (`http://www1.cse.wustl.edu/~cytron/cs531/Resources/Papers/valnum.pdf`).

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 2) 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.

```
#include "llvm/Analysis/ValueTracking.h"

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

Listing 2: Code for Pass 2.

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.

3.4 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, named `writeup.pdf`, containing the Andrew ID's of all members in your group.
- Well-commented source code for your passes (`DeadCodeElimination` and `LICM`), and associated `Makefiles`.
- A `README` file describing how to build and run your passes.
- Any tests used for verification of your code.

4 Homework Questions

4.1 Pointer Analysis (Purple Dragon Book 12.4 - 12.6, Lecture 13)

Recall that pointer analysis can be context sensitive or insensitive. Analyse the given program to obtain the object(s) the parameter `l` to the function `addFront` may point to at the start of the function. (For heap-allocated memory, denote a location as $Heap_X$, where X is the line number below where the corresponding `malloc` call occurs.)

```
 1 struct node { node* next; }
 2
 3 node* addFront(node* l, unsigned int n) {
 4     if (n == 0) return l;
 5     else {
 6         node* newnode = malloc(sizeof(node));
 7         newnode->next = l;
 8         return addFront(newnode, n - 1);
 9     }
10 }
11
12 void fun(unsigned int m) {
13     node* list = malloc(sizeof(node));
14     node* longlist = addFront(list, m);
15 }
16
17 int main() {
18     int n;
19     scanf("%d", &n);
20     fun(n);
21 }
```

Listing 3: Code for Question 4.1.

1. *Context insensitive* analysis - Called from line 8: $l \implies$
2. *Context insensitive* analysis - Called from line 14: $l \implies$
3. *Context sensitive* analysis - Called from line 8: $l \implies$
4. *Context sensitive* analysis - Called from line 14: $l \implies$

In the context sensitive analysis above, the context is the caller function. There are other kinds of contexts one may consider. One example is value sensitivity, where the context is an abstract representation of the values of the arguments of the function call. Compute the set of object(s) `longlist` might point to based on whether the argument `m` to `fun` is zero or non-zero.

1. (`m = 0`) - `longlist` $\implies$
2. (`m > 0`) - `longlist` $\implies$

Recall that pointer analysis can also be flow insensitive, flow sensitive, or path sensitive. For the code below, show the result of performing each type of pointer analysis for the dereference of pointer "p" at line 12.

```
 1  p = malloc();
 2  q = malloc();
 3  p = q;
 4  r = malloc();
 5  q = r;
 6  if(a)
 7        p = q;
 8  if(b)
 9        p = r;
10  if(!a)
11        p = malloc();
12   = *p;
```

Listing 4: Code for Question 4.1.

1. *Flow insensitive* analysis: $p_{12} \implies$
2. *Flow sensitive* analysis: $p_{12} \implies$
3. *Path sensitive* analysis: $p_{12} \implies$

Now convert the above program into SSA and compute using a flow insensitive analysis the heap locations the SSA variable corresponding to p would point to at line 12. Do the results match with either the flow sensitive analysis above? If yes, why do we need flow sensitivity at all? Why not just use the SSA version of the program? If no, state what is different?
(Hint: Are variables the only entities that can be used to store references to memory?)

4.2 Register Allocation (Purple Dragon Book 8.8, Lecture 12)

Suppose that you have a processor with five 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).

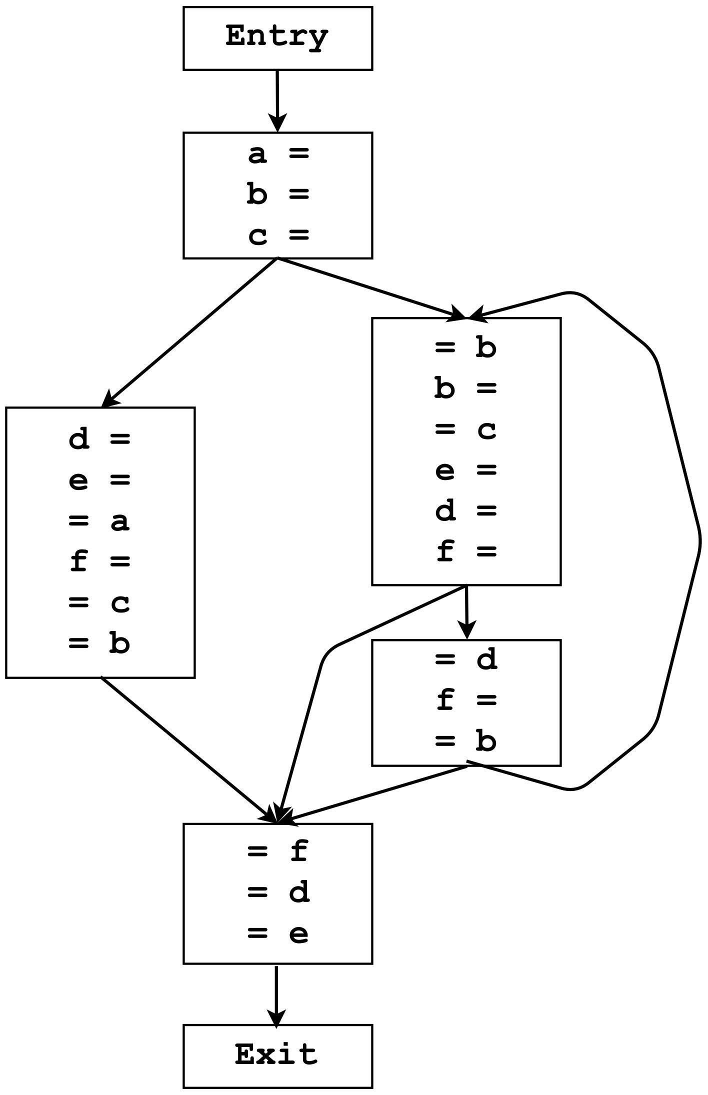

Figure 1: Code for Question 4.2.

4.3 Submission

Please submit a single PDF file on Canvas containing your solutions to the homework problems.

5 Appendix

5.1 Dominators: Example Input

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

Listing 5: Example input code.

```
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 6: Example simplified input bitcode.

5.2 Dominators: Example Output

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

Figure 2: Example analysis output.