15-745: Optimizing Compilers
Assignment 3 Writeup
Ben Humberston, bhumbers
Prashanth Suresh, psuresh

Q1: Profiling with LLVM

No work to show

Q2: Loop Invariant Code Motion

Structure:
The LoopInvariantCodeMotion pass is implemented as a *FunctionPass* rather than a *LoopPass* so that we may compute reaching definitions a single time over the whole function, then reuse the results in each loop pass. We compute reaching definitions using a corrected and improved version of our code from Assignment 2.

As in a normal *LoopPass*, loops are processed in order from highest to lowest nesting level to ensure that deeply nested invariant statements may "bubble out". Processing per loop is split into several distinct subtasks, as outlined in the algorithm for LICM:

- *computeDominance():* Runs a dataflow pass to compute the block dominance information
- *computeImmediateDominance():* Finds immediate dominance relations based on the dominance dataflow results
- *computeCodeMotionCandidateStatements()*: Marks all possible statements (instructions) which may be moved by LICM
- *applyMotionToCandidates()*: Actually mutates the program, moving allowed LICM candidates to the preheader of the loop

Note that the additional invariance checks described in the assignment writeup are necessary for the following reasons:

- *isSafeToSpeculativelyExecute(I):* This prevents moving code which may cause exceptions (like dividing by 0) or which has other side effects, like branches.
- *!I→mayReadFromMemory():* If an instruction reads from memory, then we can't know statically what value it will read . This will occur, for example, with pointers and array indexing. To be safe, we must assume that the instruction is not loop invariant because of the value it reads.

Usage:
See README. or simply execute "./RUN_ME.sh" from the ClassicalDataflow directory to first compile the passes & benchmarks and then run each of the 3 benchmarks.

The benchmarks include *basic_main, nested_main, and double_nested_main*. A C file for each is found in the "tests" directory; running the "build_and_dis_tests.sh" script followed by the "do_licm_tests.sh" script will recreate our results (or, again, simply run RUN_ME.sh).

Benchmark Results

Benchmark	**Original Performance** (Dynamic Instructions)	**Optimized Performance** (Dynamic Instructions)
basic_main.c	1,475	1,265
nested_main.c	864,255	740,795
double_nested_main.c	5,431,905	5,308,445*

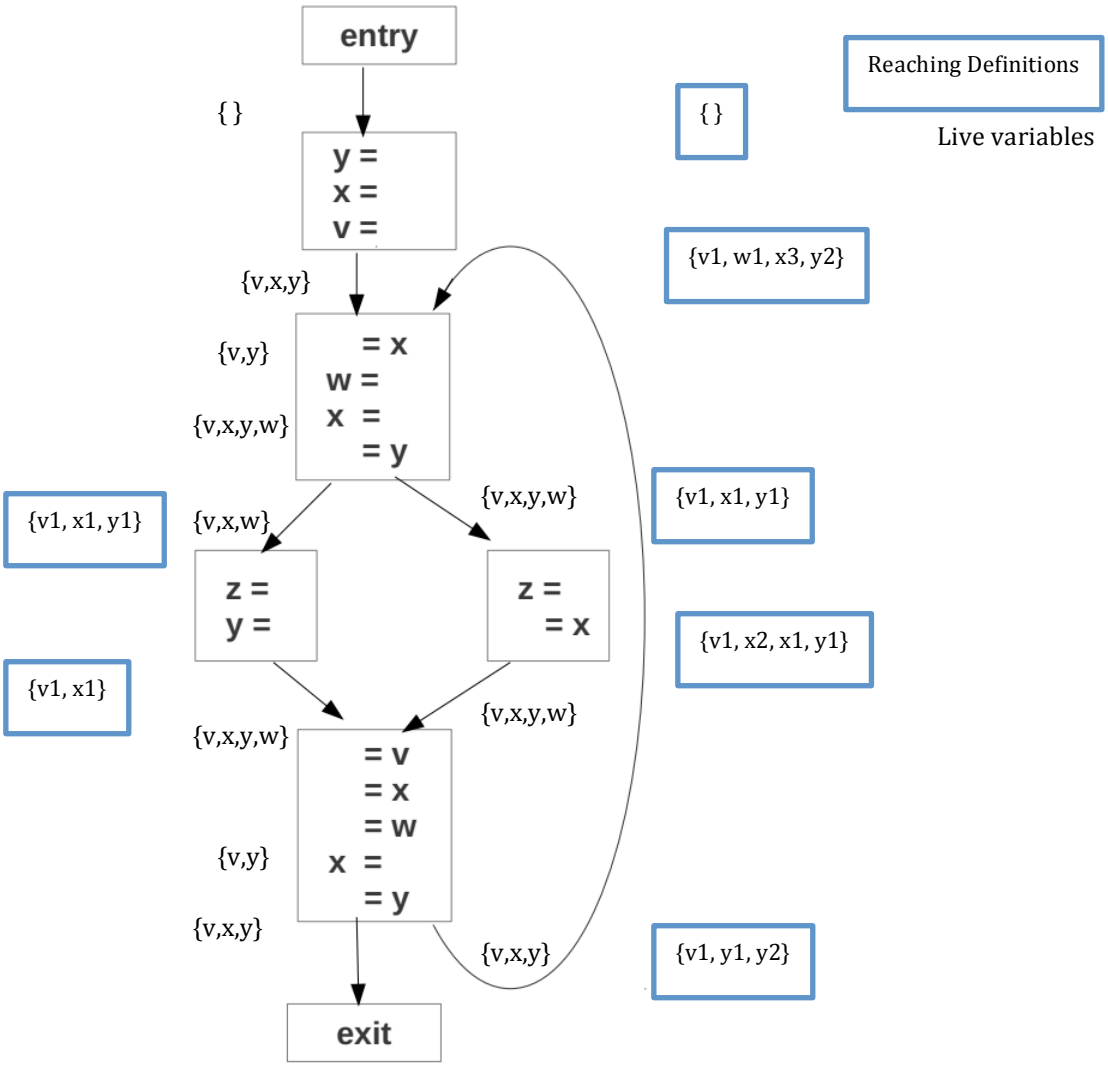
**Our LICM implementation appeared not to fully optimize this doubly-nested loop, hence giving only about the same reduction in dynamic instructions as the simpler nested_main.c benchmark.*

Q2.1: Dead Code Elimination

We chose not to implement this pass due to time constraints.

Q3: Questions

3.1 Register Allocation

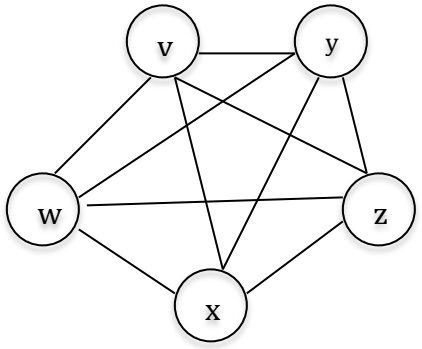


Form live range- eliminate equivalent definitions {y1,y2}

Next step: Interference graph
2 live ranges overlap.
Hence we have 5 nodes for v,x,y,z,w.

Now assume **z is alive at the end of the program**.

Links for the interference graph is drawn for the variables, which are live concurrently.



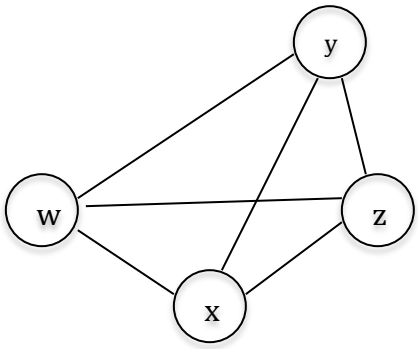
Since the variables are 5 and the register count is only 4, spilling is necessary for live range to not overlap.
Hence Chaitin coloring and spilling algorithm is used.

Step 1: If node with less than 4 neighbors present, place node on stack.
Here all the nodes of the interference graph has 4 neighbors. Hence no node can be removed.

Step 2: else node with highest degree to cost ratio is spilled.
Cost= Number of uses/definitions.
Cost of v=1+1=2
Cost of w=1+1=2
Cost of x=3+2=5
Cost of y=2+2=4
Cost of z=2

Degree to cost ratio for v=2/4=0.5
Degree to cost ratio for w=2/4=0.5
Degree to cost ratio for x=5/4=1.25
Degree to cost ratio for y=4/4=1
Degree to cost ratio for z=2/4=0.5

Hence v has the highest degree to cost ratio. Hence v is spilled.



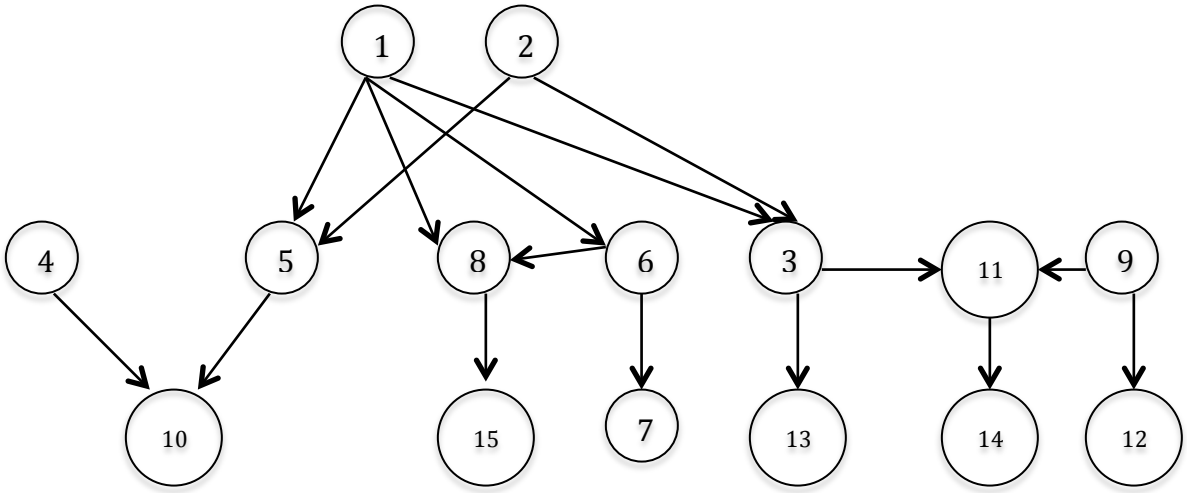
Now the modified interference graph is shown.
Step3: Insert the remaining nodes on stack and allocate different colors for the 4 nodes.
Hence v is spilled and registers allocated for the remaining nodes.

Note: Spilling v causes an additional read from memory for every iteration of the loop. Variable z could intuitively be a better option to spill since it has only 1 store every iteration and no uses. From question 3.2's cycle count we can say that 1 store will cost lesser cycles than a load for every iteration and this can utilize the pipeline fully. Additionally v has a definition outside the loop which will be a memory operation and can be avoided if z is spilled to memory and registers are allocated for v, w, x and y.

Thus the 4 registers according to the Chaitin spilling algorithm are allocated to the variables w, x, y and z.

3.2 Instruction Scheduling

1.List Scheduling –Forward Analysis

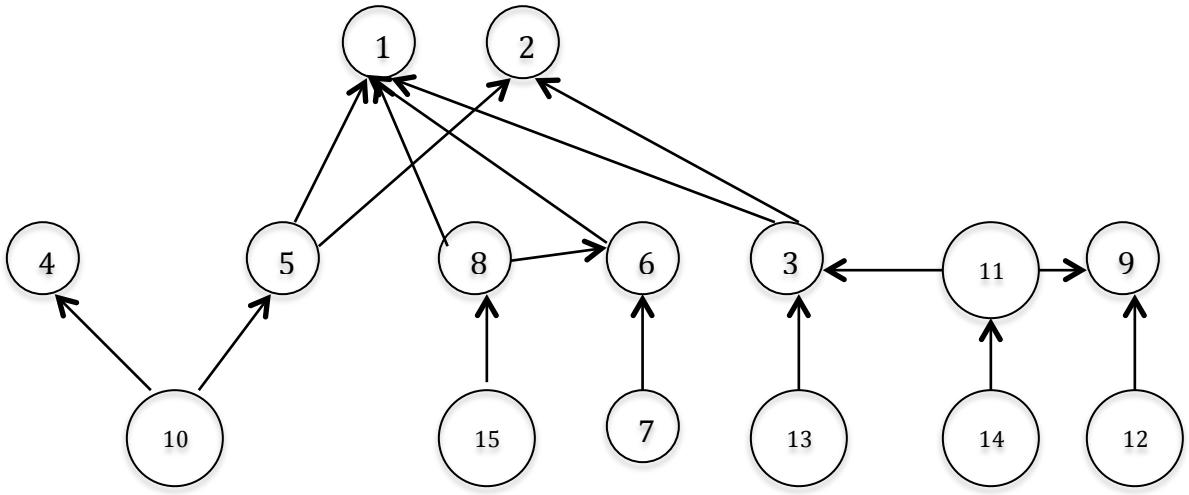


Priority Table

Instruction No	Priority
1	14
2	14
3	3
4	7
5	11
6	3
7	1
8	2
9	3
10	4
11	2
12	1
13	1
14	1
15	1

Cycle No	Ready List	Inflight	I0	I2	L/S	Comments
0	1,2,4,9	1	-	-	1	Add load instruction 1 to inflight list
1	2,4,9	1,2	-	-	2	Add load instruction 2 to inflight list
2	4,9	1,2,4	-	-	4	Add load instruction 4 to inflight list
3	9,6	4,2,6,9	6	-	9	Add load instruction 9 & 6 to inflight list
4	5,3,8,7	9,4,5,3	5	3	-	Add 5,3 to inflight list
5	8,7,13	5,9,8,7	8	-	7	Add 8,7 to inflight list
6	11,12,13	5,11,12	11	-	12	Add 11,12(store) to inflight list
7	13,14	5,13	-	-	13	Add store 13 to inflight list
8	14	5,14	-	-	14	Add store 14 to inflight list.
9	-	5	-	-	-	Wait for div to complete.
10	-	5	-	-	-	
11	-	5	-	-	-	
12	10	10	10	-	-	Add 10 to inflight list.
13	-	10	-	-	-	
14	-	10	-	-	-	
15	-	10	-	-	-	Complete

2.Backward Analysis



Priority Table

Instruction No	Priority
1	3
2	3
3	4
4	3
5	10
6	4
7	5
8	5
9	3
10	14
11	5
12	14
13	5
14	6
15	6

Cycle No	Ready List	Inflight	I0	I2	L/S	Comments
0	10,14,15,7,13,12	10,14	10	-	14	Add instruction 10,14 to inflight list
1	15,7,13,12,11	10,15	-	-	15	Add instruction 15(load) to inflight
2	7,8,11,12,13	10,7,8	8	-	7	Add 7(load), 8 to inflight list
3	11,13,12,6	10,11,13	11	-	13	Add 11, 13(load) to inflight list
4	12,6,3,4,5	5,3	5	3	-	Add 5,3 to inflight list
5	12,6,4	5,6,12	6	-	12	Add 6,12 to inflight list
6	4,9	5,4	-	-	4	Add 4(load) to inflight list
7	9	5,4,9	-	-	9	Add 9(load) to inflight list
8	-	5,4,9	-	-	-	Add store 14 to inflight list.
9	-	5,9	-	-	-	Wait for div to complete.
10	-	5	-	-	-	
11	1,2	2	-	-	1	Add 1(load) to inflight list
12	2	1,2	10	-	2	Add 2(load) to inflight list.
13	-	1,2	-	-	-	
14	-	1,2	-	-	-	
15	-	1,2	-	-	-	Complete