Writeup

By junhanz vigneshb

Analysis Passes

Dominators

This path is just implemented according to the given algorithm. When back in assignment 2 when doing the dataflow framework I did it with analyzing at the instruction level, so when here dealing with basic blocks I had to modify (simplify) the framework to deal on the basic block level.

Loop Invariant Code Motion

I initially did this with a branched landing pad branching to if the loop would be executed at all or not, but then the phi instructions in the cond part of the loop were having values used not dominating problems. So now instead I just moved the invariant instruction directly to the given preheader and let the phi instructions in the cond part of the loop to deal with if it thinks the instruction have been seen before or not and this proved to be an effective way of implementing it.
The given checks provided in the handout as: for the speculatively execute check when there yields an error if the instruction is moved the place of occurrence of the error is wrong and therefore harder for the programmer to debug their code. The may read from memory check is necessary as memory is very unpredictable of the access pattern, there might also be special cases where the memory is a memory mapped IO or alike that when touched does something special, like when reading an interrupt flag on a device might automatically clear the flag making the device ready for a further interrupt.
The benchmark I tested is as follows (all first using the mem2reg and then loop-simplify pass):

Original File	Unoptimized Count	Optimized Count
test.c	9415	7442
stest.c	259	259
duploop.c	9500	7486

Showing that it moved instructions out of the loop when possible and even when impossible it doesn't add any instruction cycles.

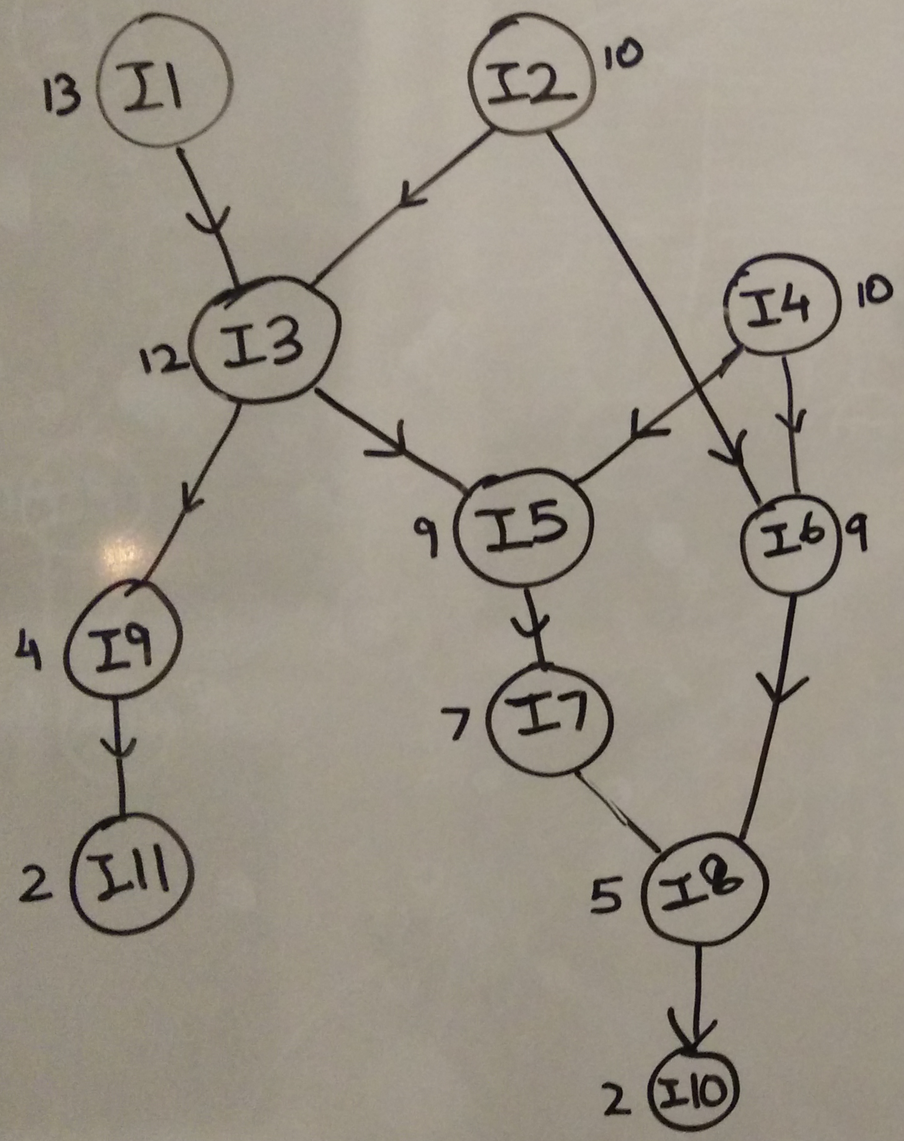
Dead Code Elimination

This implementation is pretty straightforward, locating a dead code as a code with no uses and isn't one of the cases given in the handout. The result is as follows:

Original File	Unoptimized Count	Optimized Count
dup.c	15	4
duploop.c	9500	9499

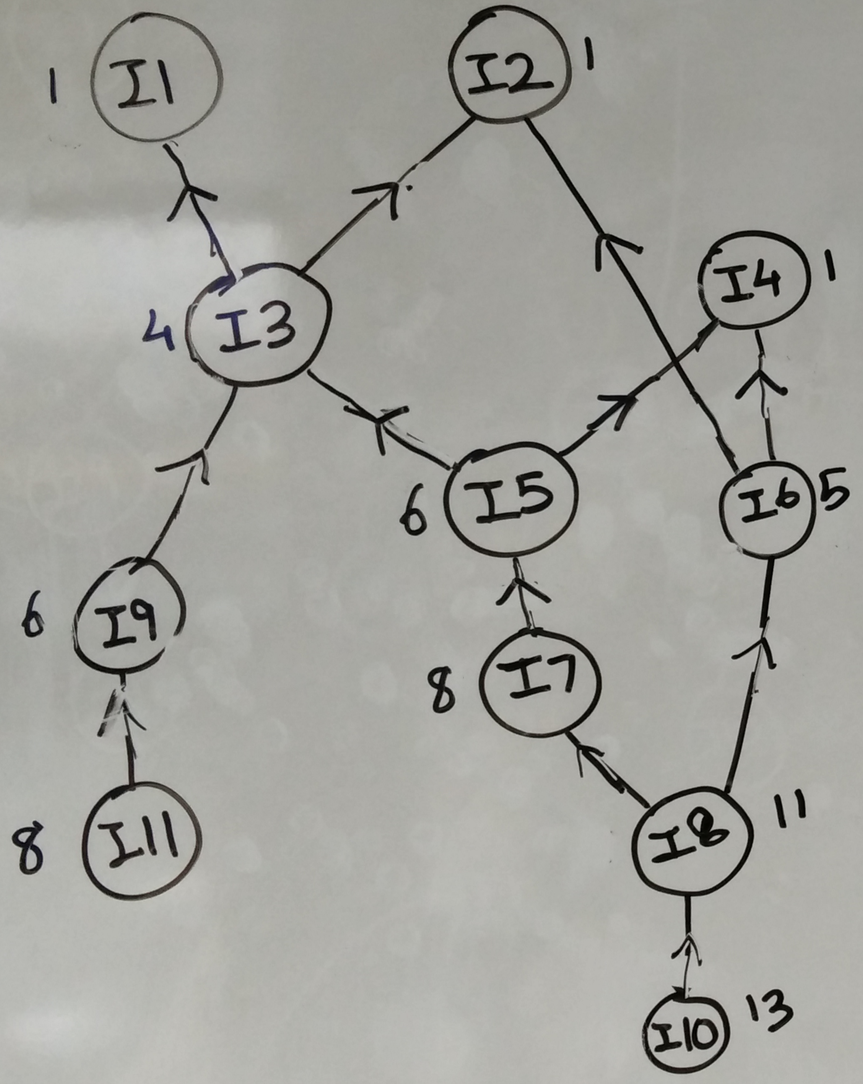
Showing that it's excellent at locating pure dead code, but when it comes to loops where a useless value updates itself during a loop it can't pick it up.

Forward List Scheduling



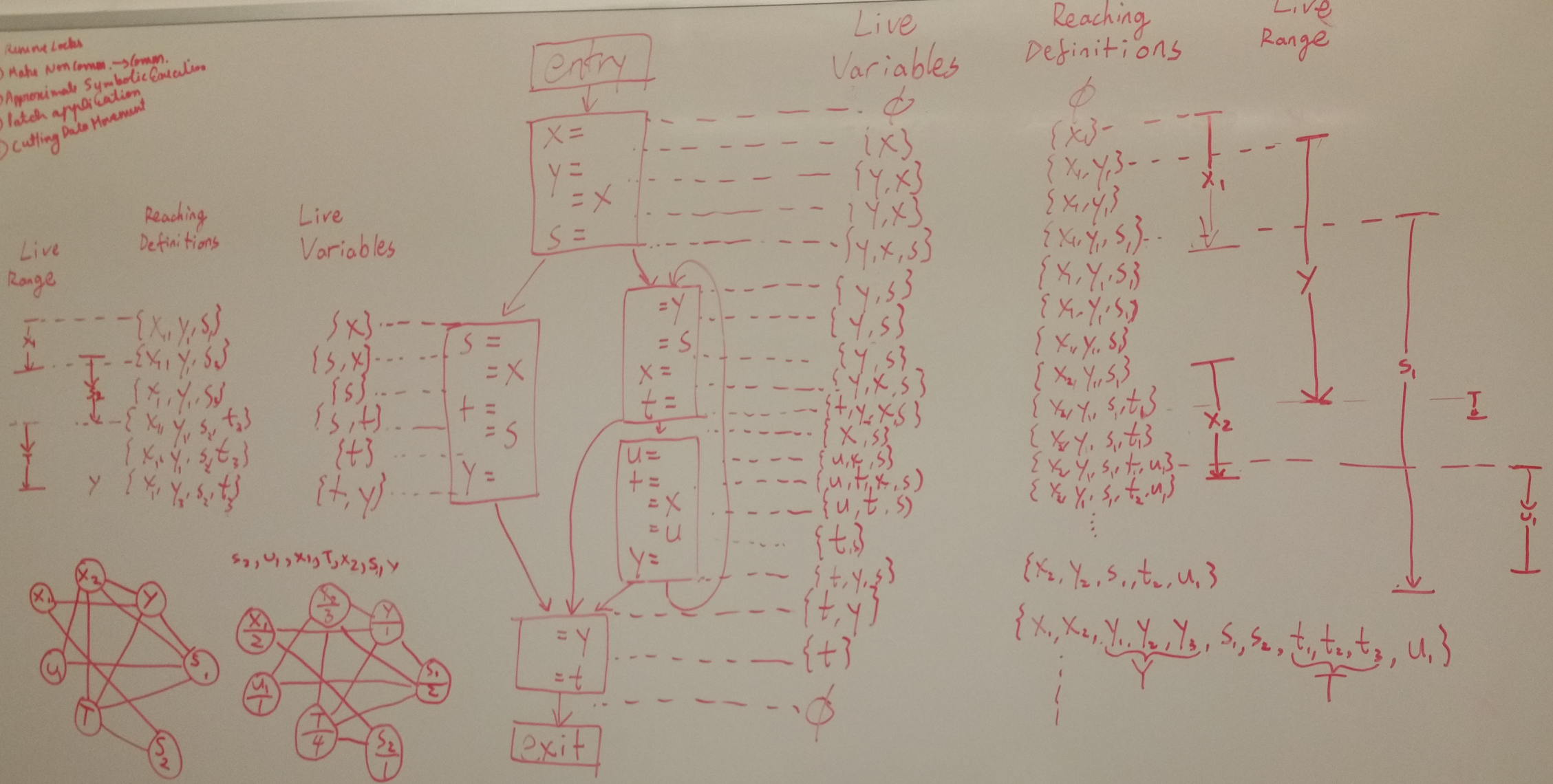
Cycle	ALU0	ALU1	LSU	RDY
0	–	–	I1	I1, I3, I4
1	–	–	I2	I2, I4
2	I3	–	I4	I3, I4
3	I3	I6	—	I6
4	I3	I6	—	{}
5	I5	I6	—	I9, I5
6	I5	I6	—	I9
7	–	I9	I7	I9, I7, I8
8	–	I9	I7	{}
9	I8	–	I11	I8, I11
10	I8	–	I11	{}
11	I8	–	–	
12	–	–	I10	I10
13	–	–	I10	

Backward List Scheduling



Cycle	ALU0	ALU1	LSU	TO BE COMPLETED
0	–	–	I1	I1
1	–	–	I2	I2,I1
2	I3	–	–	{}
3	I3	–	–	{}
4	I3	–	I4	I3,I4
5	I5	–	–	{}
6	I5	–	–	I5
7	I6	–	I7	{}
8	I6	I9	I7	I7,I6
9	I8	I9	–	I9,
10	I8	–	I11	{}
11	I8	–	I11	I8,I11
12	–	–	I10	I11
13	–	–	I10	I10,I11

1. Remove Locks
2. Make Non-Common → Common.
3. Approximate Symbolic Execution
4. Patch application
5. Cutting Data Movement



Interference Graph

Final Colored Graph