Assignment 3

Michael Choquette, Rokhini Prabhu
mchoquet, rokhinip
March 7, 2015

Implementation Notes

The loop-invariant code motion transformation works according to the following steps:

1. Compute the nearest dominating ancestor of every block, using two instances of DFS.
2. Extract all candidate instructions from the loop, where a candidate instruction is all of the following:
    - Safe to speculatively execute.
    - Not a memory read.
    - Not a phi node.
    - Not a landing pad instruction.
3. Repeatedly loop through the list of candidates, extracting everything that is either not defined in the loop or only has loop-invariant operands. Stop when a pass through the candidates list generates no new instructions.
4. Loop through the generated list, and move all of them to the loop preheater.

Some important notes:

- As per the piazza post, we do not guarantee to only move instructions that dominate all loop exits.
- Because of the order in which we examined candidates, the instructions do not need to be topologically sorted before moving them to the preheater.

The dead code elimination pass effectively implemented the faintness analysis from Homework 1. The one notable thing with this is that since the transfer function is piecewise and requires information in the input bitvector, we needed to modify parts of our dataflow framework in order to accomodate that. This in particular meant that since this function didn't compose as easy as the transfer functions for liveness, we have to create a separate `ComposedTransferFunction` class which explicitly did the compositions of transfer functions for a sequence of instructions.

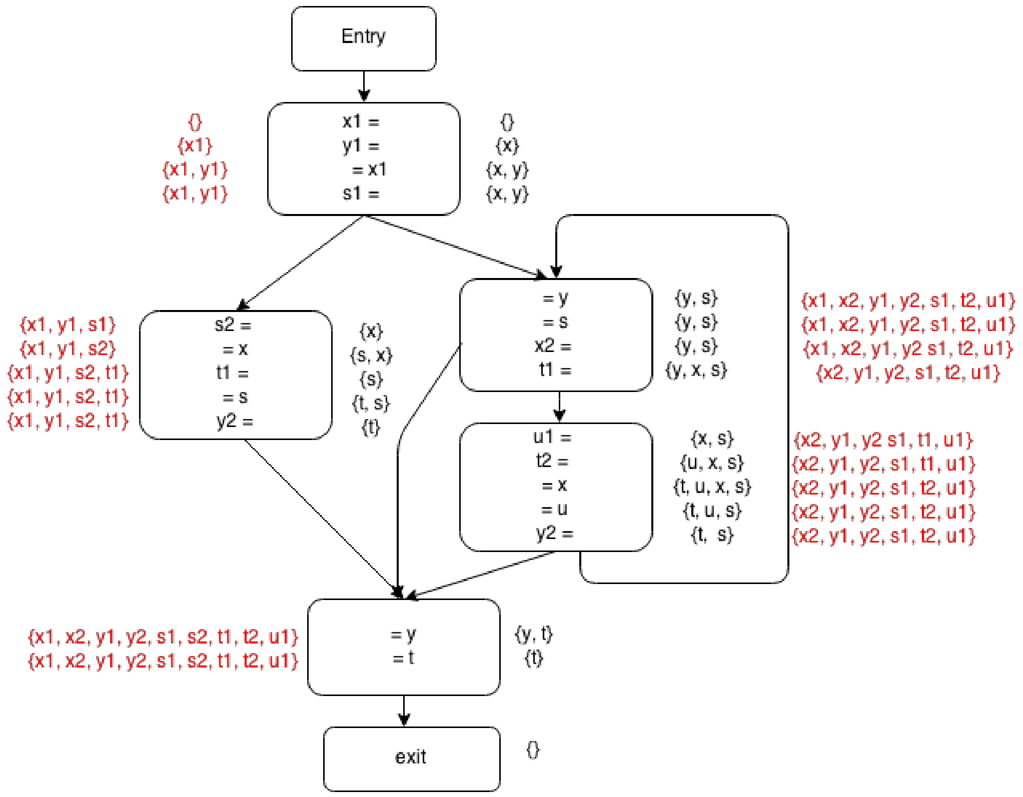
We made sure that an instruction which meet any of the following criteria is not dead code eliminated. `isa<TerminatorInst>(I)`, `isa<DbgInfoIntrinsic>(I)`, `isa<LandingPadInst>(I)` or if `I->mayHaveSideEffects()`. In addition, any operand used by these instructions is marked as not being faint so that the preceding calculations are completed successfully.

Benchmark	Instructions exe before LICM	Instructions exe after LICM
init2d.c	9762016	7017516
loop.c	4013989	3011993
zmean.c	330006	310008

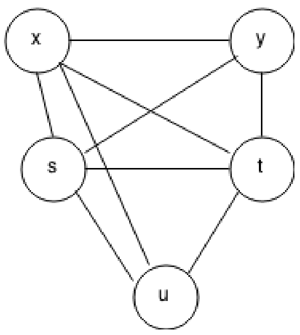
Benchmark	Instructions exe before DCE	Instructions exe after DCE
bench1.c	70006	60006
bench2.c	90006	60006

Register Allocation

- Compute the liveness and reaching definitions of the variables in the graph and compute the merged live ranges. Live ranges are in black, reaching definitions in red



- Create the interference graph:



- Perform coloring and spilling. We are using Chaitin's algorithm here. The stack we obtain is: $[t, s, y, x, u]$. We then pop the stack and assign registers as follows: t gets r0, s gets r1, y gets r2, x gets r3 and u gets r2. We see here that we are able to assign all variables to registers without spilling.

Instruction Scheduling

The answers to parts 1 and 2 are in the following table:

Clock Cycle	Ready(forwards)	Issued(forwards)	Ready(backwards)	Issued(backwards)
0	(1, 2, 4)	(1)	(1, 2, 4)	(1)
1	(2, 4)	(2)	(2, 4)	(2)
2	(4)	(4)	(4)	()
3	()	()	(4)	(4)
4	(3, 5)	(3, 5)	(3, 5)	(5)
5	(6, 7, 9, 10)	(6, 7)	(3, 7, 9)	(7)
6	(8, 9, 10)	(9, 10)	(3, 6, 8, 9, 10)	(6, 10)
7	(8)	(8)	(3, 8, 9)	(8, 9)
8	(12)	(12)	(3)	(3)
9	()	()	(12)	()
10	(11, 13)	(11, 13)	(11, 12)	(12)
11	(14)	(14)	(11, 13)	(11, 13)
12	(15)	(15)	(14)	(14)
13			(15)	(15)