Implementation of LICM pass:

Briefly, I firstly find all loop invariants and store the address of them in a vector object, then loop through all these stored invariants and move the qualified ones to the loop pre-header. Below are the details of the steps I did.

(1) Find all loop invariants.

Since the code is in SSA form, an instruction is loop invariant if its operands are either constants, coming from outside of the loop (which can be checked using Loop::contains()) or loop invariants. Therefore, I just loop through all the instructions repeatedly and check for every instruction if it satisfies the above conditions until there is no more loop invariants I can find. The reason why this step must be repeated is that the set of the found invariants is growing, so I might find new invariants which operands are in this set.

(2) Test all the found loop invariants with some conditions and move the qualified ones to the loop pre-header.

The conditions needed to check are:

- isSafeToSpeculativelyExecute(I)

This function basically help us make sure the motion of the instruction will not cause code semantic error. According to the LLVM documentation, this function checks if the instruction does not have any effects nor undefined behavior. For example, it can detect dividing by zero, loading from an invalid pointer, control flow related instructions such as terminators and PHI instructions etc.

- ! I->mayReadFromMemory()

If the instruction might read from memory, then we cannot be sure that the content it will read is available in memory if moving it to the pre-header.

- ! isa < LandingPadInst > (I)

We need the landing pad instruction(s) to ensure the pre-header executes only if it enters the loop, so we cannot move it to the pre-header.

- Check if the instruction dominates all exits a

Just need to test the instruction with all exiting blocks inside the loop because the loop-simplify pass will ensure there are no other predecessors outside the loop for the loop successors. Use DominatorTreeWrapperPass to get dominance information and use Loop::getExitingBlocks() to get all exiting blocks.

To move the qualified ones to the loop pre-header:

- Loop::getLoopPreheader() to get the loop pre-header
- Insert the invariants to be just before the loop pre-header's terminator instruction by invariant->moveBefore(preHeader->getTerminator()); Move the invariants to the loop pre-header in the same order that I added them to the set (the vector object). This will make sure I move the operands first then the entire instruction so there won't be any syntax or semantic error.

Which one of the loop structures can LICM work naturally?

Only the do-while loop can be handled naturally because only it can guarantee the loop body is executed at least once. For the other types of loops, we have to transform them into do-while loops.

How do you handle cases where there are nested loop structures?

According to the Loop Pass documentation, the Loop Pass will be executed on inner loops first then on outer loops, so I don't need to do anything special as that's the ideal order of handling them. If any loop invariant in any inner loop has been moved to the inner loop's pre-header, the pass can still consider it as a LICM candidate when processing the outer loops.

3.1

Flow Insensitive : {S1, S2, S4, S11}

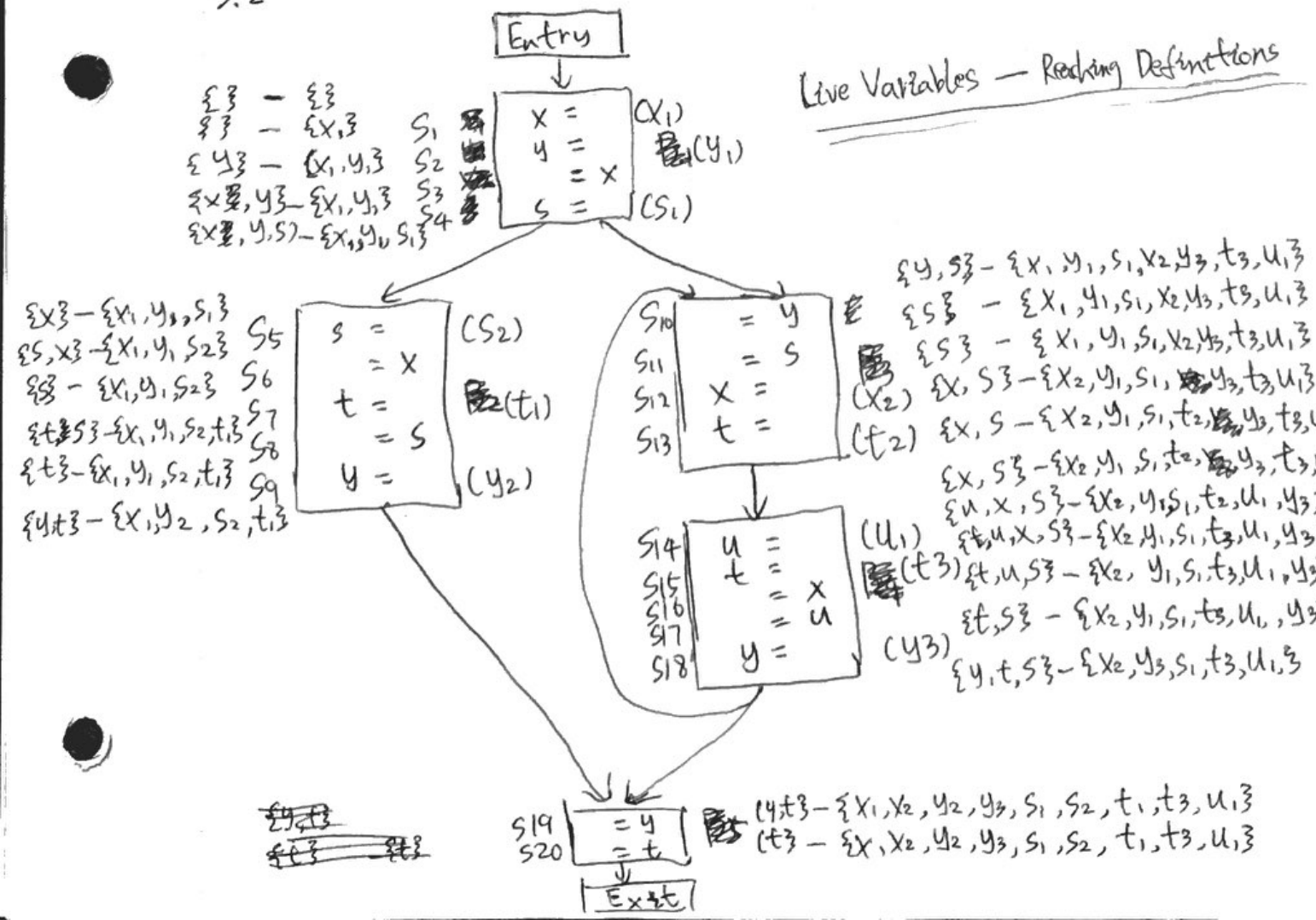
Flow Sensitive: {S2, S4, S11}

Path Sensitive: {S2, S4, S11}

Context Insensitive: {a, b}

Context Sensitive: Called from S9 : {a}
Called from S12 : {b}
Called from S5 : {a}

3.2



~~Live Variables:~~

Live Ranges:

x: s_3, s_4, s_5, s_6, s_{12}, s_{13}, s_{14}, s_{15}, s_{16}

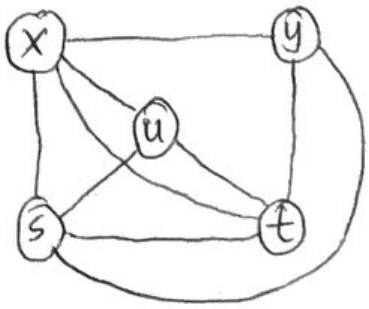
y: s_2, s_3, s_4, s_9, s_{10}, s_{18}, s_{19}

s: s_4, s_5, s_6, s_7, s_{10}, s_{11}, s_{12}, s_{13}, s_{14}, s_{15}, s_{16}, s_{17}, s_{18}

t: s_7, s_8, s_9, ~~s_{15}, s_{16}, s_{17}, s_{12}~~ s_{15}, s_{16}, s_{17}, s_{19}, s_{20}

u: s_{15}, s_{16}

Interference graph:



Coloring:

~~t~~ u, y → Red

x → Yellow

s → Blue

t → Green

4 ~~to~~ colorable