**15-745: Optimizing Compilers**

**Assignment 2 Writeup**

**Ben Humberston, bhumbers**

**Prashanth Suresh, psuresh**

**Implementation Summary**

**Generic Data Flow Framework:**

See DataFlow.h and .cpp. Rather than use function pointers passed into a dataflow function, we define the generic dataflow as an abstract interface. Any pass which wishes to use the framework must subclass *DataFlow* and implement the *applyMeet()* and *applyTransfer()* functions with the logic for the pass. Calling *run()* then executes the dataflow analysis and returns the values for each block’s IN and OUT.

Values (sets) are represented by BitVector, and we decided that using a vector of *Value* pointers as the domain type was sufficiently generic to cover most passes. The results of calling *run()* includes the mapping from input domain elements to positions in the value bit vectors (this is equivalent to the order of entries in the given domain, but it seemed prudent to make this mapping explicit).

**Liveness:**

See liveness.c. The generic dataflow framework is used to compute the liveness of each variable at the IN and OUT of all blocks (see “*LivenessDataFlow*” implementation in liveness.cpp for meet operator and transfer function). Then, these liveness values are expanded to the interior of each block by iterating backward over instructions and updating liveness instruction-by-instruction.

In order to make liveness results sensible, we have special handling for nodes. If not otherwise alive, the liveness of any operands used in instructions is only propagated backward along the edge to the predecessor block associated with that operand.

**Reaching Definitions:**

The implementation of this pass is similar in many respects to the liveness pass, except that analysis moves forward rather than backward and the transfer function is modified to use thecorrect gen and kill sets for reaching definitions. There is also no special node handling necessary in this pass. It should be noted, then, that all definitions reach the exit block as a result of the SSA form.

**Code Listing**

* DataFlow.h & .cpp
* liveness.cpp
* reaching\_definitions.cpp
* /tests/\*.c *(tests)*

**Test Listing**

Run “./ClassicalDataflow/RUN\_ ME.sh” in a bash shell in order to 1) build the passes & test cases and 2) run all test cases. Test output will be shown in the console.

All tests are executed using both the liveness and reaching definition passes.

* sum.c
  + The example specified in the assignment writeup. As shown in the sample output below, our tests did not produce exactly the same IR as given in the writeup, but the pass results are largely comparable.
* basic.c
  + Tests absolute basic cases as well as nested conditionals and function calls (though this last was hard to verify due to tail recursion optimizations from LLVM).
* loc\_exp\_usage.c
  + Tests whether locally exposed uses are correctly processed

**Test Output**

(only showing sum.c test output as an example)

**Liveness**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* LIVENESS OUTPUT FOR FUNCTION: sum \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Domain of values: {%a | %b | %cmp4 | %res.06 | %i.05 | %mul | %inc | %exitcond | %res.0.lcssa}

define i32 @sum(i32 %a, i32 %b) #0

entry:

Liveness: {%a | %b}

%cmp4 = icmp slt i32 %a, %b

Liveness: {%a | %b | %cmp4}

br i1 %cmp4, label %for.body, label %for.end

Liveness: {%a | %b}

for.body:

%res.06 = phi i32 [ %mul, %for.body ], [ 1, %entry ]

%i.05 = phi i32 [ %inc, %for.body ], [ %a, %entry ]

Liveness: {%b | %res.06 | %i.05}

%mul = mul nsw i32 %res.06, %i.05

Liveness: {%b | %i.05 | %mul}

%inc = add nsw i32 %i.05, 1

Liveness: {%b | %mul | %inc}

%exitcond = icmp eq i32 %inc, %b

Liveness: {%b | %mul | %inc | %exitcond}

br i1 %exitcond, label %for.end, label %for.body

Liveness: {%b | %mul | %inc}

for.end:

%res.0.lcssa = phi i32 [ 1, %entry ], [ %mul, %for.body ]

Liveness: {%res.0.lcssa}

ret i32 %res.0.lcssa

Liveness: {}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* END LIVENESS OUTPUT FOR FUNCTION: sum \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Reaching Definitions:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* REACHING DEFINITIONS OUTPUT FOR FUNCTION: sum \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Domain of values: {i32 %a | i32 %b | %cmp4 = icmp slt i32 %a, %b | %res.06 = phi i32 [ %mul, %for.body ], [ 1, %entry ] | %i.05 = phi i32 [ %inc, %for.body ], [ %a, %entry ] | %mul = mul nsw i32 %res.06, %i.05 | %inc = add nsw i32 %i.05, 1 | %exitcond = icmp eq i32 %inc, %b | %res.0.lcssa = phi i32 [ 1, %entry ], [ %mul, %for.body ]}

define i32 @sum(i32 %a, i32 %b) #0

entry:

Reaching Defs: {i32 %a | i32 %b}

%cmp4 = icmp slt i32 %a, %b

Reaching Defs: {i32 %a | i32 %b | %cmp4 = icmp slt i32 %a, %b}

br i1 %cmp4, label %for.body, label %for.end

Reaching Defs: {i32 %a | i32 %b | %cmp4 = icmp slt i32 %a, %b}

for.body:

Reaching Defs: {i32 %a | i32 %b | %cmp4 = icmp slt i32 %a, %b | %res.06 = phi i32 [ %mul, %for.body ], [ 1, %entry ] | %i.05 = phi i32 [ %inc, %for.body ], [ %a, %entry ] | %mul = mul nsw i32 %res.06, %i.05 | %inc = add nsw i32 %i.05, 1 | %exitcond = icmp eq i32 %inc, %b}

%res.06 = phi i32 [ %mul, %for.body ], [ 1, %entry ]

%i.05 = phi i32 [ %inc, %for.body ], [ %a, %entry ]

%mul = mul nsw i32 %res.06, %i.05

Reaching Defs: {i32 %a | i32 %b | %cmp4 = icmp slt i32 %a, %b | %res.06 = phi i32 [ %mul, %for.body ], [ 1, %entry ] | %i.05 = phi i32 [ %inc, %for.body ], [ %a, %entry ] | %mul = mul nsw i32 %res.06, %i.05 | %inc = add nsw i32 %i.05, 1 | %exitcond = icmp eq i32 %inc, %b}

%inc = add nsw i32 %i.05, 1

Reaching Defs: {i32 %a | i32 %b | %cmp4 = icmp slt i32 %a, %b | %res.06 = phi i32 [ %mul, %for.body ], [ 1, %entry ] | %i.05 = phi i32 [ %inc, %for.body ], [ %a, %entry ] | %mul = mul nsw i32 %res.06, %i.05 | %inc = add nsw i32 %i.05, 1 | %exitcond = icmp eq i32 %inc, %b}

%exitcond = icmp eq i32 %inc, %b

Reaching Defs: {i32 %a | i32 %b | %cmp4 = icmp slt i32 %a, %b | %res.06 = phi i32 [ %mul, %for.body ], [ 1, %entry ] | %i.05 = phi i32 [ %inc, %for.body ], [ %a, %entry ] | %mul = mul nsw i32 %res.06, %i.05 | %inc = add nsw i32 %i.05, 1 | %exitcond = icmp eq i32 %inc, %b}

br i1 %exitcond, label %for.end, label %for.body

Reaching Defs: {i32 %a | i32 %b | %cmp4 = icmp slt i32 %a, %b | %res.06 = phi i32 [ %mul, %for.body ], [ 1, %entry ] | %i.05 = phi i32 [ %inc, %for.body ], [ %a, %entry ] | %mul = mul nsw i32 %res.06, %i.05 | %inc = add nsw i32 %i.05, 1 | %exitcond = icmp eq i32 %inc, %b}

for.end:

Reaching Defs: {i32 %a | i32 %b | %cmp4 = icmp slt i32 %a, %b | %res.06 = phi i32 [ %mul, %for.body ], [ 1, %entry ] | %i.05 = phi i32 [ %inc, %for.body ], [ %a, %entry ] | %mul = mul nsw i32 %res.06, %i.05 | %inc = add nsw i32 %i.05, 1 | %exitcond = icmp eq i32 %inc, %b}

%res.0.lcssa = phi i32 [ 1, %entry ], [ %mul, %for.body ]

ret i32 %res.0.lcssa

Reaching Defs: {i32 %a | i32 %b | %cmp4 = icmp slt i32 %a, %b | %res.06 = phi i32 [ %mul, %for.body ], [ 1, %entry ] | %i.05 = phi i32 [ %inc, %for.body ], [ %a, %entry ] | %mul = mul nsw i32 %res.06, %i.05 | %inc = add nsw i32 %i.05, 1 | %exitcond = icmp eq i32 %inc, %b | %res.0.lcssa = phi i32 [ 1, %entry ], [ %mul, %for.body ]}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* END REACHING DEFINITION OUTPUT FOR FUNCTION: sum \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Q3**

**3.1 Lazy Code Motion**

**CFG for original code:**

(1) x=x+3

(2) if(y>5)

(3) z=z+x

(4) x=4

(6) y=y-5

(7) z=z+x

Entry

(9) Return z

Exit

**Pass 1: CFG with Anticipation**

**Values Key:** (z+x) / (x+3) / (y-5)

(1) x=x+3

(3) z=z+x

(6) y=y-5

Entry

(9) Return z

Exit

(2) if(y>5)

(7) z=z+x

(4) x=4

0/0/0

1/0/0

0/0/0

1/0/0

1/0/0

1/0/1

0/0/0

0/0/0

0/0/0

1/0/0

1/0/0

1/0/0

0/1/0

**Pass 2: Early Placement Pass**

t=z+x

z=t

(3) z=t

(6) y=y-5

Entry

(9) Return z

Exit

(2) if(y>5)

(7) z=t

(4) x=4

0

1

0

1

1

1

0

0

0

1

1

1

0

(1) x=x+3

Earliest z+x (anticipated here, but not available.)

**Pass 3: Lazy Code Motion**

Place at most postponable position.

t=z+x can be postponed to

to IN of (3), IN of (6), OUT of

(6), IN of (7).

(3) z=t

(6) y=y-5

Entry

(9) Return z

Exit

(2) if(y>5)

(7) z=t

(4) x=4

0

1

0

1

1

1

0

0

0

1

1

1

0

(1) x=x+3

Earliest z+x (anticipated here but not available.)

t=z+x

t=z+x

1

According to the lazy code motion algorithm,

z+x is moved to the most postponable

position as long as it is not used.

**Pass 4: Code Motion and Cleanup Pass**

Here, the temporary assignments

are removed and the expression

is moved to the most postponable

position without causing

redundancy (latest position).

t=z+x

(3) z=t

(6) y=y-5

Entry

(9) Return z

Exit

(2) if(y>5)

(7) z=t

(4) x=4

0

1

0

1

1

1

0

0

0

1

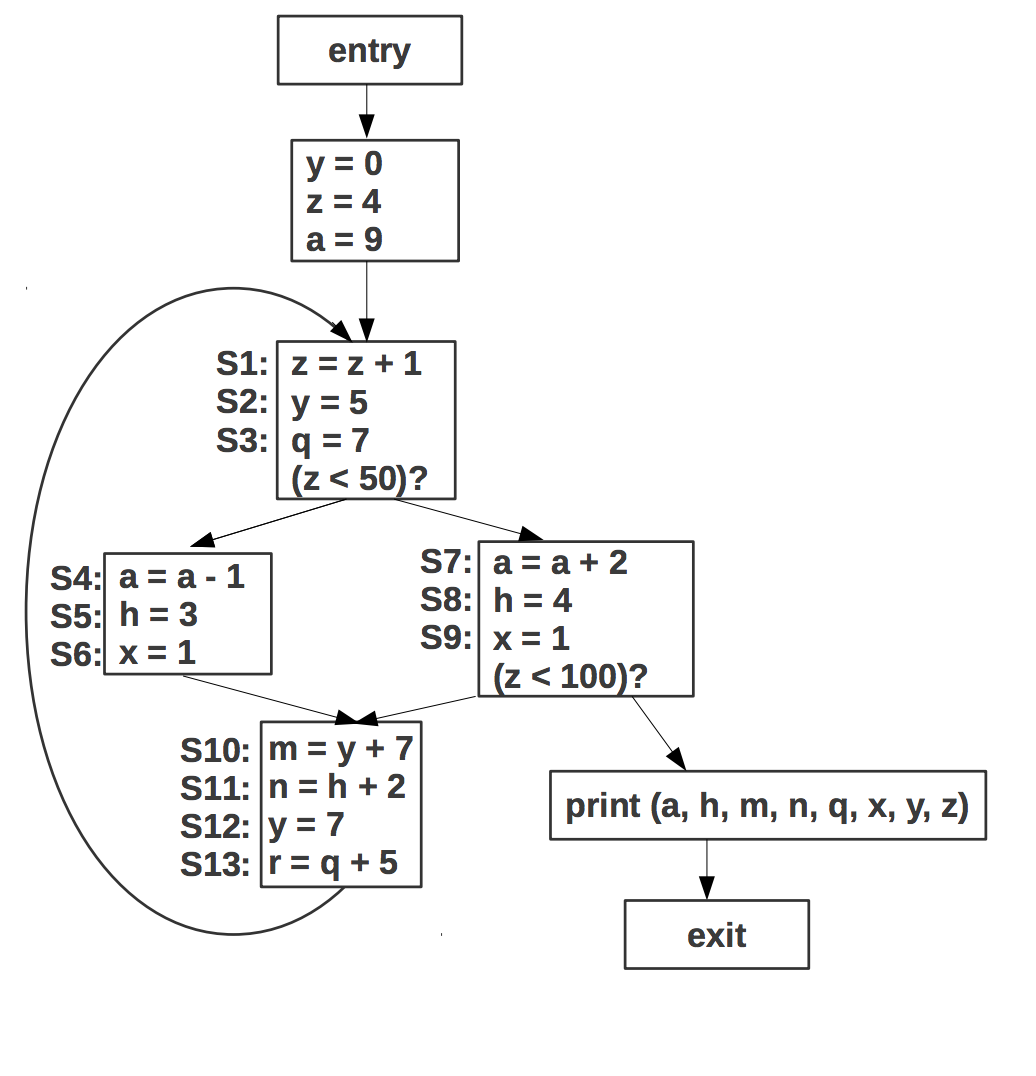
1

1

0

(1) x=x+3

**3.2 Loop Invariant Code Motion**



Loop Invariant Code Motion

Reaching Definitions:

Block 1:

IN={s1, s3, s4, s5, s6, s7, s8, s9, s10, s11, s12, s13, d1, d2, d3}

OUT={s1, s2, s3, s4, s5, s6, s7, s8, s9, s10, s11,, s13, d3}

Block2 (right):

IN={s1, s2, s3, s4, s5, s6, s7, s8, s9, s10, s11, s13, d3}

OUT={s1, s2, s3, s4, s5, s6, s10, s11, s13}

Block3 (Left):

IN={s1, s2, s3, s4, s5, s6, s7, s8, s9, s10, s11, s13, d3}

OUT={s1, s2, s3, s7, s8, s9, s10, s11, s13}

Block4:

IN={s1, s2, s3, s4, s5, s6, s7, s8, s9, s10, s11, s13}

OUT={s1, s3, s4, s5, s6, s7, s8, s9, s10, s11, s12, s13}

**(cont’d on next page)**

**3.2 (cont’d)**

Loop Invariant Instructions:

S2, S3, S10, S12, S13

S2 is a constant definition of y.

S3 is a constant definition of q.

S10 is loop invariant since reaching definition of y is invariant.

S12 is a constant definition but it can be removed since it is killed by a redefinition of y at s2 and is not used anywhere else.

S13 is loop invariant since the reaching definition of q is invariant.

Of the 6 only s2, s3, s12 and s13 can be moved to the pre-header by the loop invariant code motion pass.

S10 cannot be moved since there is a path to exit where the previous value of m is reached and is used.

**3.2 (cont’d)**

**After Lazy Code Motion Pass**

y=0

z=4

a=9

Pre-header Block

S2: y=5

S12: q=7

S13: r=q+5

Entry

S1: z=z+1

(z<50)?

S4: a=a-1

S5: h=3

S6: x=1

S7: a=a+2

S8: h=4

S9: x=1

(z<100)?

Exit

S10: m=y+7

S11: n=h+2

Print (a, h, m, n, q, x, y, z)