Final exam

Final exam will cover all material covered in course – i.e., all compiler phases

– **But Focus** will be on after-midterm material

100 minutes exam, similar in format to midterm
Optimization Frameworks
Roadmap

Last time:
– Optimization overview
  • Soundness and completeness
– Simple optimizations
  • Peephole
  • LICM

This time:
– More Optimization
– Analysis frameworks
Outline

Review Dominators
Introduce some more advanced concepts
– Static single assignment (SSA)
– Dataflow propagation
DOMINATOR REVIEW
Dominator terms

Domination (A dominates B):
– to reach block B, you must have gone through block A

Strict Domination (A strictly dominates B)
– A dominates B and A is not B

Immediate Domination (A immediately dominates B)
– A immediately dominates B if A dominates B and has no intervening dominators
Dominator example
Dominance frontier

**Definition:**
For a block $x$, the dominance frontier of $x$ is the set of nodes $Y$ such that

- $x$ dominates an immediate predecessor of $Y$
- $x$ does not strictly dominate $Y$
Example

Diagram:
- A
- B
- C
- D
- E

Connections:
- A to B
- A to C
- B to D
- C to D
- D to E
- E to A
- C to A
- B to C
STATIC SINGLE ASSIGNMENT
Goal of SSA Form

Build an intermediate representation of the program in which each variable is assigned a value in at most 1 program point:

- \( x = 1 \)
- \( z = 2 \)
- \( y = 3 \)

- \( x = y \)
- \( z = y \)
- \( w = z \)

Statically: There is at most one assignment statement that assigns to \( k \)

Dynamically: \( k \) can be assigned to multiple times

\[
i = 0;
while( i < 10)\
{k = i + 1;}
\]
Conversion

We’ll make new variables to carry over the effect of the original program

\[
\begin{align*}
  x &= 1 \\
  x &= x \\
  y &= x
\end{align*}
\]

\[
\begin{align*}
  x_1 &= 1 \\
  x_2 &= x_1 \\
  y_1 &= x_2
\end{align*}
\]
Benefits of SSA Form

There are some obvious advantages to this format for program analysis

– Easy to see the *live range* of a given variable $x$ assigned to in statement $s$
  • The region from “$x = ...;$” until the last use(s) of $x$ before $x$ is redefined
  • In SSA form, from “$x_i = ...;$” to all uses of $x_i$, e.g., “... = f(..., x_i, ...);”

– Easy to see when an assignment is *useless*
  • We have “$x_i = ...;$” and there are *no uses* of $x_i$ in any expression or assignment RHS
  • “‘$x_i = ...;' is a useless assignment”
  • “‘$x_i = ...;' is dead code”

In other words, some useful information is pre-computed, or at least easily recoverable from SSA form

*Warning 1:* Dead code = useless assignments + unreachable code
Optimizations Where SSA Helps

Dead-Code Elimination

At “if (b < 4)”, b is only reached by “b = 2;” Therefore, the else branch is unreachable (dead), and can be removed.

```c
int a = 9;
int b = 2;

if (g < 12){
    a = 1;
} else {
    if (b < 4){
        a = 2;
    } else {
        a = 3;
    }
}

b = a;
return 2;
```

```c
int a1 = 9;
int b1 = 2;

if (g1 < 12){
    a2 = 1;
} else {
    if (b1 < 4){
        a3 = 2;
    } else {
        a4 = 3;
    }
}

a5 = (a3, a4);
a6 = (a2, a5);
b2 = a6;
return 2;
```
Optimizations Where SSA Helps

Constant-propagation/constant-folding

```c
int a = 30;  // 6
int b = 9;   // (a / 5);
int c;
if (true) {  // 12
    c = 2;  // 10;
} else {  // 2
    c = b * 4;
}
if (c > 10) {  // 2
    c = c - 10;
}
return c * (60 / a);
```
What About Conditionals?

Which y to use?

- $x = 5$
- $x = x - 1$
- $x < 3$

- $y = x \times 2$
- $w = y$
- $w = x - y$
- $z = x + y$

- $y = x - 3$

- $x_1 = 5$
- $x_2 = x_1 - 1$
- $x_2 < 3$

- $y_1 = x_2 \times 2$
- $w_1 = y_1$
- $w_2 = y - x$
- $z = x + y$

- $y_2 = x_2 - 3$

What About Conditionals?
We introduce a special symbol $\Phi$ at such points of confluence
$\Phi$’s arguments are all the instances of variable $y$ that might be the most recently assigned variant of $y$
Returns the “correct” one
Do we need a $\Phi$ for $x$?
– No!
Computing Phi-Function Placement

Intuitively, we want to figure out cases where there are multiple assignments that can reach a node. To be safe, we can place a $\Phi$ function for each assignment at every node in the dominance frontier.
Pruned Phi Functions

This criterion causes a bunch of useless $\Phi$ functions to be inserted
– Cases where the result is never used “downstream” (useless)

*Pruned SSA* is a version where useless $\Phi$ nodes are suppressed
DATAFLOW ANALYSIS
Dataflow framework idea

Many analyses can be formulated as how data is transformed over the control flow graph

Propagate static information from:
– the beginning of a single basic block
– the end of a single basic block
– The join points of multiple basic blocks
Dataflow framework idea

Meet Lattice

Transfer function
– How data is propagated from one end of a basic block to the other

Meet operation
– Means of combining lattice between blocks
Dataflow analysis direction

Forward analysis
– Start at the beginning of a function’s CFG, work along the control edges

Backwards analysis
– Start at the end of a function’s CFG, work against the control edges

Continuously propagate values until there is no change
Dataflow-Analysis Example 1

Reaching definitions

Before p1: \(\{<p_1, x>\}\)
After p1: \(\{<p_1, x>\}\)

Before p2: \(\{<p_1, x>, ...\}\)
After p2: \(\{<p_2, x>, ...\}\)

Before p3: \(\{<p_2, x>, ...\}\)
After p3: \(\{<p_2, x>, <p_3, y>, ...\}\)

Transfer function:

\[
p_1: x = 1;
\]

\[
p_2: x = 2;
\]

\[
p_3: y = x;
\]

Data: sets of \(<\text{program-point}, \text{variable}>\) pairs

Note: for expository purposes, it is convenient to assume we have a statement-level CFG rather than a basic-block-level CFG.
**Dataflow-Analysis Example 1**

**Reaching definitions**

Before p1: 
After p1: \{<p1, x>\}

Before p2: \{<p1, x>, ...\} 
After p2: \{<p2, x>, ...\}

Before p3: \{<p2, x>, <p4,x>, ...\} 
After p3: \{<p2, x>, <p3, y>, <p4,x>,...\}

**Meet operation**: Union of sets (of <program-point, variable> pairs)

Before p4: 
After p4: \{<p4, x>\}

- p1: x = 1; 
- p2: x = 2 
- p3: y = x; 
- p4: x = 7; 

Note: for expository purposes, it is convenient to assume we have a statement-level CFG rather than a basic-block-level CFG.
Dataflow-Analysis Example

Reaching definitions: Why is it useful?
Answers the question “Where could this variable have been defined?”

Before p1:
After p1: {<p1, x>}
p1: x = 1;

Before p2: {<p1, x>, ...
After p2: {<p2, x>, ...
p2: x = 2;

Before p3: {<p2, x>, <p4, x>, ...
After p3: {<p2, x>, <p3, y>, <p4, x>, ...
p3: y = x;

Before p4: {<p4, x>}
p4: x = 7;
Dataflow-Analysis Example 2

Live Variables

Before p1: {x}
After p1: {x}

Before p2: {x}
After p2: {x, y}

Before p3: {x, y}
After p3: {x, y}

Before p4: {x}
After p4: {x}

Before p5: {x}
After p5: {x}

Before p6: {x}
After p6: {x}

p1: x = 1;
if (...) {
  p2: y = 0;
  p3: z = x + y;
}
p4: x = 2;
p5: z = 3;
p6: cout << x;

Transfer function:

Data: sets of variables

z is not live after p5, and thus p5 is a useless assignment (= dead code)
The end: or is it?

Covered a broad range of topics
- Some formal concepts
- Some practical concepts

What we skipped
- Linking and loading
- Interpreters
- Register allocation
- Performance analysis / Proofs