Program Analysis

Class #2

Readings

- The Program Dependence Graph and Its Use in Optimization
- Program slicing
- A Survey of Program Slicing Techniques
- Dragon book
Program Analysis

Data-Flow Analysis
(wrap up)

Recall: Dominance (algorithm)

*Input*: N, pred, entry

*Output*: domin: n → {n}

- domin(entry) = {entry}
- foreach n ∈ N - {entry}
  - domin(n) = N
- label: change = false
- foreach n ∈ N - {entry}
  - T = \( \bigcap_{b \in \text{pred}(n)} \text{domin}(p) \)
  - D = \{n\} \cup T
  - if D ≠ domin(n)
    - change = true
    - domin(n) = D
  - if change = true goto label
- return domin

---

\[ \text{CFG} \]

entry → \( S_1 \) → \( S_2 \) → \( S_3 \) → \( S_4 \) → \( S_5 \) → \( S_6 \) → exit
Dominance as a DF Problem

Local information:

- **GEN[B]**
- **KILL[B]**

By propagation:

- **IN[B]**
- **OUT[B]**

1. Initialize **IN[B]**, **OUT[B]** sets
2. Iterate over all **B** until no changes
   - On each iteration, visit all **B**, and compute
     **IN[B]**, **OUT[B]** as

\[
\text{IN}[B] = \bigcap \text{OUT}(P), \ P \in \text{pred}(B)
\]

\[
\text{OUT}[B] = \text{IN}(B) \cup (\text{GEN}[B] - \text{KILL}[B])
\]
Other DF Approaches (worklist)

Data-flow for nodes 1, 2, 3 never changes but is computed on every iteration of the algorithm.

In general, nodes involved in the computation may be a small subset of the nodes in the graph.

For example, what if I want to compute RD only for f1?

```
algorithm RDWorklist
Input: GEN[B], KILL[B] for all B
Output: reaching definitions for each B
Method:
    initialize IN[B], OUT[B] for all B; add successors of B initially involved in computation to worklist W
    repeat
        remove B from W
        Oldout=OUT[B]
        compute IN[B], OUT[B]
        if oldout != OUT[B] then add successors of B to W endif
    until W is empty
```
Other DF Approaches (worklist)

Compute RD for f1 using RDWorklist
- Add successors of 3, 10 to W
- remove 4 from W, compute IN[4], OUT[4], etc.

Program Analysis

Dependence Analysis
Dependence Analysis

Important for
• Optimization
  • Instruction scheduling
  • Data-cache optimization
• Software engineering
  • Program understanding
  • Reverse engineering
  • Debugging

Two main kinds of dependences
• Data related
• Control related

Data-Dependence Graph (DDG)

A **DDG** has one node for every variable (basic block) and one edge representing the flow of data between two nodes

Different types of data dependences
• Flow: def to use
• Anti: use to def
• Output: def to def
• Input: use to use

Further classifiable as
• Loop-carried: requires iterations
• Loop-independent: occurs anyway
**Data-Dependence Graph**

- \( X = 1 \)  
- \( Z > 1 \)

- \( X = 2 \)  
- \( Z > 2 \)

- \( Y = X + 1 \)

- \( Z = X - 3 \)  
- \( X = 4 \)

- \( Z = X + 7 \)

**Control-Dependence Analysis**

**Definition**

Let \( G \) be a CFG, with \( X \) and \( Y \) nodes in \( G \). \( Y \) is **control-dependent** on \( X \) iff

1. There exists a path \( P \) from \( X \) to \( Y \) with any \( Z \) in \( P \) (excluding \( X \) and \( Y \)) post-dominated by \( Y \) and
2. \( X \) is not post-dominated by \( Y \)

(intuitively: there are two edges out of \( X \); traversing one edge always leads to \( Y \), the other may not lead to \( Y \))
Control-Dependence Analysis

What are the control dependences for each statement in the CFG on the right?

Dependences:
- B1, exit – entry
- B2 – B1T
- B3 – B2T
- B4 – B1F
- B5 – B2F, B1F
- B6 – B2F, B1F
Computing CD Using FOW

1. Construct basic CDG
2. Add region nodes
Computing CD Using FOW

1.1 Build a CFG

1.2 Augment the CFG by adding a node Start with edge (Start, entry) labeled “T” and edge (Start, exit) labeled “F”; call this AugCFG
Computing CD Using FOW

1.1 Build a CFG

1.2 Augment the CFG by adding a node Start with edge (Start, entry) labeled “T” and edge (Start, exit) labeled “F”; call this AugCFG

1.3 Construct the postdominator tree for AugCFG
Computing CD Using FOW

1.1 Build a CFG

1.2 Augment the CFG by adding a node Start with edge (Start, entry) labeled “T” and edge (Start, exit) labeled “F”; call this AugCFG

1.3 Construct the postdominator tree for AugCFG

1.4 Consider set S of edges (m, n) in AugCFG such that n does not postdominate m
1.4 Consider set $S$ of edges $(m, n)$ in AugCFG such that $n$ does not postdominate $m$

1.5 Consider, for each edge $(m, n)$ in $S$, those nodes in the Pdom tree from $n$ to least common ancestor $L$ of $m$ and $n$

- Including $L$ if $L$ is $m$
- Excluding $L$ if $L$ is not $m$
Computing CD Using FOW

1.5 Consider, for each edge \((m, n)\) in \(S\), those nodes in the Pdom tree from \(n\) to least common ancestor \(L\) of \(m\) and \(n\)

- Including \(L\) if \(L\) is \(m\)
- Excluding \(L\) if \(L\) is not \(m\)

<table>
<thead>
<tr>
<th>Edge</th>
<th>(L)</th>
<th>Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start, En</td>
<td></td>
<td>En, 1</td>
</tr>
<tr>
<td>1, 2</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>1, 4</td>
<td>Ex</td>
<td>4, 5, 6</td>
</tr>
<tr>
<td>2, 3</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>2, 5</td>
<td>Ex</td>
<td>5, 6</td>
</tr>
</tbody>
</table>

All identified nodes are control dependent on \(m\)
1.5 Consider, for each edge 
\((m, n)\) in \(S\), those nodes in 
the Pdom tree from \(n\) to 
least common ancestor \(L\) 
of \(m\) and \(n\) 
• Including \(L\) if \(L\) is \(m\) 
• Excluding \(L\) if \(L\) is not \(m\)

<table>
<thead>
<tr>
<th>Edge</th>
<th>L</th>
<th>Nodes</th>
<th>CD on</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start, En</td>
<td>Ex</td>
<td>En, I</td>
<td>Start, T</td>
</tr>
<tr>
<td>1, 2</td>
<td>Ex</td>
<td>2</td>
<td>1, T</td>
</tr>
<tr>
<td>1, 4</td>
<td>Ex</td>
<td>4, 5, 6</td>
<td>1, F</td>
</tr>
<tr>
<td>2, 3</td>
<td>Ex</td>
<td>3</td>
<td>2, T</td>
</tr>
<tr>
<td>2, 5</td>
<td>Ex</td>
<td>5, 6</td>
<td>2, F</td>
</tr>
</tbody>
</table>

All identified nodes are control dependent on \(m\)

Why is the set we obtain correct?

1.6 Create the actual CDG

<table>
<thead>
<tr>
<th>Edge</th>
<th>L</th>
<th>Nodes</th>
<th>CD on</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start, En</td>
<td>Ex</td>
<td>En, I</td>
<td>Start, T</td>
</tr>
<tr>
<td>1, 2</td>
<td>Ex</td>
<td>2</td>
<td>1, T</td>
</tr>
<tr>
<td>1, 4</td>
<td>Ex</td>
<td>4, 5, 6</td>
<td>1, F</td>
</tr>
<tr>
<td>2, 3</td>
<td>Ex</td>
<td>3</td>
<td>2, T</td>
</tr>
<tr>
<td>2, 5</td>
<td>Ex</td>
<td>5, 6</td>
<td>2, F</td>
</tr>
</tbody>
</table>
Program-Dependence Graph (PDG)

A PDG for a program P is the combination of the data- and control- dependence graphs for P.

A PDG contains nodes representing statements in P and edges representing either control or data dependence between nodes.

PDG exercise

Compute the PDG for the program on the left:

1  x = 1;
2  y = 2;
3  if (c)
4      x++;
5  while(d) {
6      x--;
7      y += 1;
8  }
9  printf("%d", y);
Program Analysis

Slicing

Program Slicing

- Slicing overview
- Types of slices, levels of slices
- Methods for computing slices
- Interprocedural slicing
Some History

1. Mark Weiser, 1981
   Experimented with programmers to show that slices are:

   “The mental abstraction people make when they are debugging a program” [Weiser]

   Used Data Flow Equations

2. Ottenstein & Ottenstein – PDG, 1984

Applications

- Debugging
- Program Comprehension
- Reverse Engineering
- Program Testing
- Measuring Program Metrics
  - Coverage, Overlap, Clustering
- Refactoring
Static VS Dynamic Slicing

Static Slicing
- Statically available information only
- No assumptions on input
- Computed slice in general inaccurate
  ➞ Approximations ➞ results may be useless

Dynamic Slicing
- Computed on a given input
- Typically more precise ➞ more useful for applications such as debugging and testing

Types of Slicing (backward)

A **backward slice** of a program with respect to a program point \( p \) and set of program variables \( V \) consists of all statements and predicates in the program that may affect the value of variables in \( V \) at \( p \)

The program point \( p \) and the variables \( V \) together form the **slicing criterion**, usually written \( <p, V> \)
Types of Slicing (backward)

General approach: backward traversal of program flow
- Slicing starts from point $p$ ($C = (p, V)$)
- Examines statements that are executed before $p$ ($\neq$ statements that appear before $p$)
- Adds statements that affect value of $V$ at $p$, or execution of $p$
- Transitive dependences are considered

Types of Slicing (backward)

Criterion <10, product>

1. read (n)
2. i := 1
3. sum := 0
4. product := 1
5. while $i \leq n$ do
6. sum := sum + i
7. product := product * i
8. $i := i + 1$
9. write (sum)
10. write (product)
Types of Slicing (backward)

Criterion <10, product>

1. read (n)
2. i := 1
3. sum := 0
4. product := 1
5. while i <= n do
6.   sum := sum + i
7.   product := product * i
8.   i := i + 1
9. write (sum)
10. write (product)

Types of Slicing (executable)

A slice is executable if the statements in the slice form a syntactically correct program that can be executed.

If the slice is computed correctly (safely), running the executable slice produces the same result for variables in V at p for all inputs.
Types of Slicing (executable)

Criterion <10, product>

1. read (n)
2. i := 1
3. sum := 0
4. product := 1
5. while i <= n do
6. sum := sum + i
7. product := product * i
8. i := i + 1
9. write (sum)
10. write (product)
Types of Slicing (executable)

Criterion $<10, \text{product}>$

read (n)
i := 1

product := 1
while i <= n do

    product := product * i
    i := i + 1

write (product)

Types of Slicing (forward)

A **forward slice** of a program with respect to a program point $p$ and set of program variables $V$ consists of all statements and predicates in the program that may be affected the value of variables in $V$ at $p$.

The program point $p$ and the variables $V$ together form the **slicing criterion**, usually written $<p, V>$.
Types of Slicing (forward)

General approach: forward traversal of program flow
- Slicing starts from \( p \ (C = (p, V)) \)
- Examine all statements that are executed after \( p \)
- Keep statements that are affected by the values of \( V \) at \( p \) or by the execution of \( p \)
- Transitive dependences considered
- Shows downstream code that depend on a specific variable or statement
- Can show the code affected by a modification

1. read (n)
2. \( i := 1 \)
3. \( \text{sum} := 0 \)
4. \( \text{product} := 1 \)
5. while \( i <= n \) do
6. \( \text{sum} := \text{sum} + i \)
7. \( \text{product} := \text{product} * i \)
8. \( i := i + 1 \)
9. write (sum)
10. write (product)
Types of Slicing (forward)

Criterion <3, sum>

1. read (n)  
2. i := 1  
3. sum := 0  
4. product := 1  
5. while i <= n do  
6. sum := sum + i  
7. product := product * i  
8. i := i + 1  
9. write (sum)  
10. write (product)

Types of Slicing (forward)

Criterion <1, n>

1. read (n)  
2. i := 1  
3. sum := 0  
4. product := 1  
5. while i <= n do  
6. sum := sum + i  
7. product := product * i  
8. i := i + 1  
9. write (sum)  
10. write (product)
Types of Slicing (forward)

Criterion <1, n>

1. read (n)
2. i := 1
3. sum := 0
4. product := 1
5. while i <= n do
   6. sum := sum + i
   7. product := product * i
   8. i := i + 1
9. write (sum)
10. write (product)

Methods for Computing Slices

Data-flow on the flow graph
- Intraprocedural: CFG
- Interprocedural: ICFG

Reachability on a dependence graph
- Intraprocedural: program-dependence graph (PDG)
- Interprocedural: system-dependence graph (SDG)
Methods for Computing Slices

Data-flow on the flow graph
- Intraprocedural: CFG
- Interprocedural: ICFG

Reachability on a dependence graph
- Intraprocedural: program-dependence graph (PDG)
- Interprocedural: system-dependence graph (SDG)

Slicing as a reachability on the PDG
- Slicing criterion: node n in the PDG
- Simple method: slice = all nodes from which n is reachable
- Backward traversal of the PDG
Slicing as a reachability on the PDG

What is the slice for <y, 8> ?

```
1 x = 1;
2 y = 2;
3 if (c)
4   x++;
5 while(d) {
6     x--;
7     y += 1;
7 }
8 printf("%d", y);
```

Now let’s do it using the PDG
Slicing as a reachability on the PDG

What is the slice for <y, 8>?

Slice = backward reachable nodes

1. x = 1;
2. y = 2;
3. if (c)
   4. x++; 
   5. while(d) {
       6. x--; 
       7. y += 1; 
     }
8. printf("%d", y);