SSA (wrap-up) & Dependence Analysis

CS2210
Lecture 16

CFG

1
i := 1
j := 1
k := 0

2
k < 100

3
j < 20

4
return j

5
j := i
k := k + 1

6
j := k
k := k + 2

7
Dominator Tree

![Dominator Tree Diagram]

Dominance Frontiers

<table>
<thead>
<tr>
<th>node n</th>
<th>DF(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{}</td>
</tr>
<tr>
<td>2</td>
<td>{2}</td>
</tr>
<tr>
<td>3</td>
<td>{2}</td>
</tr>
<tr>
<td>4</td>
<td>{}</td>
</tr>
<tr>
<td>5</td>
<td>{7}</td>
</tr>
<tr>
<td>6</td>
<td>{7}</td>
</tr>
<tr>
<td>7</td>
<td>{2}</td>
</tr>
</tbody>
</table>
Iterated Dominance Frontiers

- For i: $DF^1(\{1\}) = DF(\{1\}) = \{}$
  - no phi functions for i
- For k,j:
  $DF^1(\{1,5,6\}) = DF(\{1,5,6\}) = \{7\}$
  $DF^2(\{1,5,6\}) = DF(\{1,5,6\} \cup DF^1(\{1,5,6\})) = DF(\{1,5,6,7\}) = \{2,7\}$
  $DF^3(\{1,5,6\}) = DF(\{1,5,6\} \cup DF^2(\{1,5,6\})) = DF(\{1,5,6,7,2\}) = \{2,7\} = DF^+(\{1,5,6\})$
  - phi functions in blocks 2 and 7

Phi-Function Insertion
Variable Renaming

Constant Propagation with SSA form

- $W =$ list of all statements in the SSA program, (c always means some constant)
- While $W$ not empty remove some statement $S$ from $W$
  - if $S$ is $x = \phi(c, c, c, ..., c)$ replace it by $x = c$
  - if $S$ is $x = c$ delete $S$
    - for each statement $T$ that uses $x$ substitute $c$ for $x$ in $T$
    - $W = W \cup \{T\}$
Example

delete $i_1 = 1$
replace $j_3 = i_1$ with $j_3 = 1$
(also replaces uses of $j_1, k_1$)

Other Modifications

- Copy propagation
  - single argument $x := \phi(y)$ $\phi$-function or a copy assignment $x := y$ can be substituted for every use of $x$
Conditional Constant Propagation

- In the example, if $j = 1$ always then block 6 will never execute, so $j$ will stay 1
- if block 6 is executed $k$ is assigned to $j$ and block 6 will be executed since $j$ will be $> 20$ eventually

---

Standard Lattice

- $V[v] = T$ have not seen assignment to $v$ executed
  - $V[v] = 4$ have seen that $v = 4$ is executed but never any other value assigned
  - $V[v] = \bot$ have seen at least two values or a a value not predictable at compile time (e.g., $v = \text{read()}$)
Add Control Flow Tracking

- E(B) = false, have seen no evidence of B being executed
- E(B) = true, seen evidence that B can be executed
- Initially all variables start with T and all blocks B with E[B] = false

Constraints (1)

- Any variable v with no definition (input), formal parameter to a procedure, or uninitialized variable must have V[v] = ⊥
- Start block E[B_1] = true (is executable)
- for every executable block with one successor C set E[C] = true
- For any executable assignment v := x op y where V[x] = c_1 and V[y] = c_2, set V[v] = c_1 op c_2
- For any assignment x = x op y, where V[x] = T or V[y] = ⊥, set V[v] = ⊥
Constraints (2)

- For any executable assignment \( v := \phi(x_1, \ldots, x_n) \), where \( V[x_i] = c_i \), \( V[x_j] = c_2 \), \( c_1 \neq c_2 \) and the \( i \)-th and \( j \)-th predecessors are executable, set \( V[v] = \bot \).

- For any executable assignment \( v := \text{Mem()} \) or \( x := \text{call()} \), set \( V[v] = \bot \).

- For any executable assignment \( v := \phi(x_1, \ldots, x_n) \), where \( V[x_i] = \bot \) and the \( i \)-th predecessor is executable, set \( V[v] = \bot \).

Constraints (3)

- For any assignment \( v := \phi(x_1, \ldots, x_n) \) and \( V[x_i] = c \), and for every \( j \) either the \( j \)-th predecessor is not executable, or \( V[x_j] = \top \), or \( V[x_j] = c \) set \( V[v] = c \).

- For any executable branch if \( (x < y) \) goto L1 else L2 where \( V[x] = c_1 \) and \( V[y] = c_2 \) set \( E[L1] = \text{true} \) or \( E[L2] = \text{true} \) depending on \( c_1 < c_2 \).

- For any executable branch if \( (x < y) \) goto L1 else L2 where \( V[x] = \bot \) or \( V[y] = \bot \) set \( E[L1] = \text{true} \) and \( E[L2] = \text{true} \).
Algorithm

- Use two worklist $W_v$ for variables and $W_B$ for blocks
- Pick a variable from $W_v$ and process the constraints (4-9), or pick an executable block from $W_B$
- Whenever a new block is marked executable, it and the executable successors are added to the worklist $W_B$
- Whenever a variable’s valuation changes, it is added to the worklist $W_v$

Example

![Diagram showing the example with variables and blocks and their transitions]

CS2210 Compiler Design 2004/5
Dependence Analysis

Dependence Analysis

- Computes the data and control dependences
- Important for
  - Instruction scheduling (for ILP)
  - Loop parallelization
  - Data cache optimization
    - By loop reordering
Dependence Kinds

- Control dependence
  - S1 \& S2
- True (or flow)
  - S1 \& S2
- Anti-dependence
  - S1 \& S2
- Output dependence
  - S1 \& S2
- Input “dependence”
  - S1 \& S2 - two statements (instructions) read the same input

Dependence DAG

- A directed acyclic graph
  - Nodes are instructions / statements
  - Edges represent dependences

Example:

1. \texttt{r3 := [r15](4)}
2. \texttt{r4 := [r15+4](4)}
3. \texttt{r2 := r3 - r4}
4. \texttt{r5 := [r12](4)}
5. \texttt{r12 := r12 + 4}
6. \texttt{r6 := r3 * r5}
7. \texttt{[r15+4](4) := r3}
8. \texttt{r5 := r6 + 2}
Instruction Schedules

- To find feasible schedule need to know latencies
  - Latency(I_1, n_1, I_2, n_2) := number cycles latency incurred by beginning execution of I_2’s n_2^{th} cycle while executing cycle n_1 of I_1
  - Conflict(I_1, I_2) = true if I_1 must precede I_2 for correct execution else false

Resource Vectors

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>add.s</td>
<td>U</td>
<td>S,A</td>
<td>A,R</td>
<td>R,S</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mul.s</td>
<td>U</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>N</td>
<td>N,A</td>
<td>R</td>
</tr>
</tbody>
</table>
Computing Latency

- Match up resource vectors so that there is no conflict (structural hazard)
  - Add up total cycles
- Simple, but slow
- Faster alternative: DFAs!
  - DFA states encode set of resources in use
  - Transition on instruction only possible if resources available
    - Fraser & Proebsting pioneered this
    - Improved by Müller and by Bala & Rubin

Dependences in Loops

- Goal
  - Data cache optimization for array accesses
    - Determine whether reordering of array accesses is legal (= does not change semantics)
  - Loop parallelization
    - Execute array accesses out of order (but obeying data dependences)
Counting Loop Definitions

- A loop is in **canonical form** iff its index runs from 1 to N by increments of 1
- Loops are **perfectly nested** if they are nested and only the innermost loop has statements other than for statements
- The **iteration space** of k perfectly nested loops is the k-dimensional polyhedron consisting of all k-tuples of the loop index values
  - a **Iteration space traversal** = sequence of vectors of index values encountered in loop execution (aka lexicographic enumeration ≤)

Array Variable Dependences

- Dependences are a function of
  - Index variables
  - Statements
- Statement sequence: $S_1 \prec S_2$ extended for array statements:
  $$S_1[i_1, ..., i_k] \prec S_2[i_1', ..., i_k'] asds$$
  - This means that either
    - $S_1$ precedes $S_2$ (in loop body) and
      $<i_1, ..., i_k> \leq <i_1', ..., i_k'>$ OR
    - $S_1$ follows (or is equal to) $S_2$ and
      $<i_1, ..., i_k> < <i_1', ..., i_k'>$
Example

\[
\begin{align*}
\text{for } i_1 & := 1 \text{ to } 3 \text{ do } \\
& \text{for } i_2 := 1 \text{ to } 4 \text{ do } \\
\text{S1: } & t := x + y \\
\text{S2: } & a[i_1,i_2] := b[i_1,i_2] + c[i_1,i_2] \\
\text{S3: } & b[i_1,i_2] := a[i_1,i_2-1] \ast d[i_1+1,i_2] \\
\end{align*}
\]

Sequences
- S2[i1,i2-1] \text{ } \delta \text{ } S3[i1,i2]
- S2[i1,i2] \text{ } \delta \text{ } S3[i1,i2]

Dependences
- S2[i1,i2-1] \delta^f S3[i1,i2]
- S2[i1,i2] \delta^a S3[i1,i2]

Vectors

- **Distance Vector:**
  \[ d = \langle d_1, \ldots, d_k \rangle \]
  means that iteration
  \[ \langle i_1 + d_1, \ldots, i_k + d_k \rangle \]
  depends on iteration
  \[ \langle i_1, \ldots, i_k \rangle \]

- **Direction vector:**
  = approximation of distance vector(s)
  Elements are integer ranges or unions of ranges of the form \([0,0], [1, \infty], [-\infty,-1], [-\infty, \infty]\)
Dependence Vectors

- \( d = \langle [d_1^-, d_1^+], \ldots, [d_k^-, d_k^+] \rangle \) where 
  \( d_i^- \in \mathbb{Z} \cup \{-\infty\}, d_i^+ \in \mathbb{Z} \cup \{\infty\} \)
- Loop-independent and loop-carry dependences
  - \( \delta_0 \) independent
  - \( \delta_i \) carried by loop at nesting level \( i \)
- \( S_2[i_1, i_2-1] \delta_0, S_3[i_1, i_2] \)
- \( S_2[i_1, i_2] \delta_v^n, S_3[i_1, i_2] \)
- \( S_2: a[i_1, i_2] := b[i_1, i_2] + c[i_1, i_2] \)
- \( S_3: b[i_1, i_2] := a[i_1, i_2-1]^* d[i_1+1, i_2] \)
- Distance vector notation:
  - \( S_2[i_1, i_2-1] <0,1>, S_3[i_1, i_2] \)
  - \( S_2[i_1, i_2] <0,0>, S_3[i_1, i_2] \)

Dependence Testing

- Determine dependences between loop iterations
  - E.g., to reorder statements, parallelize
  - Diophantine equations:
    - \( 2*i_1 + 1 = 3*i_2 - 5 \)
    - Constrained: \( 1 <= i_{1,2} <= 4 \)
    - Solution in general NP
      - So only sufficient conditions for non-dependence

---

CS2210 Compiler Design 2004/5
Dependence Test

for $i_1 := 1$ to $h_1$
  for $i_2 := 1$ to $h_2$
    ...
      for $i_n := 1$ to $h_n$
        ...
          $x[\ldots, a_0 + a_1 * i_1 + \ldots + a_n * i_n]$
        ...
      endfor
    ...
  endfor
endfor

GCD test:

If

- $\gcd(\bigcup_{i=1}^{n} \text{sep}(a_j, b_j, j)) | \sum_{j=0}^{n}(a_j - b_j)$

then references to $x$ are independent

$\text{sep}(a_j, b_j, j) = \{a-b\}$ if direction of $j$

is = else $\{a, b\}$

For $a[3*i-5]$ and $a[2*i+1]$:

- $\gcd(3-2) | (-5 -1 + 3 -2)$ may have same iteration dependence
- $\gcd(3, 2) | -5$ may have inter-iteration dependence

But not $\gcd(4-2) | (-1+4-2)$

And not $\gcd(4, 2) | 1$

- Must have independence

GCD Test Example

- $\gcd(3-2) | (-5 -1 + 3 -2)$ may have same iteration dependence
- $\gcd(3, 2) | -5$ may have inter-iteration dependence

But not $\gcd(4-2) | (-1+4-2)$

And not $\gcd(4, 2) | 1$

Must have independence

for $i := 1$ to 4 do
  $b[j] = a[3*i-5]+2.0$
  $a[2*i+1] := 1.0/i$
endfor

for $i := 1$ to 4 do
  $b[j] = a[4*i]+2.0$
  $a[2*i+1] := 1.0/i$
endfor
Separability

- Two array references are \textit{separable} :=
  in each pair of subscript positions the expressions are
  of the form \( a_i j + b_1 \) and \( a_i j + b_2 \)
- \textit{Weakly separable} := \( a_1 i j + b_1 \) and \( a_2 i j + b_2 \)
- For separable references dependence testing is trivial, a dependence exist if
  - \( a=0 \) and \( b_1 = b_2 \)
  - \( (b_1 - b_2) | a \leq h_i j \)
- For weakly separable references more complicated conditions

More Dependence Tests

- More powerful dependence tests have been developed as GCD is quite weak
  - Extended GCD
  - Omega
- Exploit more sophisticated mathematical techniques to avoid more false positives
  - Used commonly in parallelizing compilers
  - Important for scientific codes
Alias Analysis

- Determine whether a storage location may be accessed in more than one way
- Example in C:
  
  ```c
  p = & x; x and *p are aliases for the same location
  ```
- Other sources of aliasing:
  - ?

Memory Disambiguation

- Want to **disambiguate** memory accesses for optimization:
  ```c
  exam1() {
  int a,k;
  extern int* q;
  ... 
  k = a+5;
  f(a,&k);
  *q = 13;
  k = a+5;/* is this redundant? */
  ...
  }
  ```
### Alias Information Classification

<table>
<thead>
<tr>
<th></th>
<th>May Aliasing</th>
<th>Must aliasing</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Flow-sensitive:</strong></td>
<td>if (d) p = &amp; x; {{x,*p}}</td>
<td>if (d) p = &amp; x; {{x,*p}}</td>
</tr>
<tr>
<td></td>
<td>else</td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>p = &amp; y; {{y,*p}}</td>
<td>p = &amp; y; {{y,*p}}</td>
</tr>
<tr>
<td></td>
<td>L: {{x,*p},{y,*p}}</td>
<td>L: {}</td>
</tr>
<tr>
<td><strong>Flow-insensitive:</strong></td>
<td>if (d) p = &amp; x; {{x,*p},{y,*p}}</td>
<td>const int *p = &amp; x;</td>
</tr>
<tr>
<td></td>
<td>else</td>
<td>{{x,*p}}</td>
</tr>
<tr>
<td></td>
<td>p = &amp; y; {{x,*p},{y,*p}}</td>
<td>no other assignments</td>
</tr>
<tr>
<td></td>
<td>L: {{x,*p},{y,*p}}</td>
<td>to p throughout program</td>
</tr>
</tbody>
</table>

### Steensgaard’s Algorithm

- A flow-insensitive algorithm for C
- O(n α(n,n)) complexity
  - Near linear, n number of statements
- Computes **points-to sets**
  - Pts(p) = {x,y} p may point to variables x and y
  - May information (flow insensitive)
Alias-relevant Statements

- \( x = y \)
- \( x = \& y \)
- \( x = \star y \)
- \( x = \text{op}(y_1, \ldots, y_n) \)
- \( x = \text{allocate}(y) \)
- \( \star x = y \)
- \( x = \text{fun}(f_1, \ldots, f_n) \rightarrow (r_1, \ldots, r_m) S^* \)
- \( x_1, \ldots, x_m = p(y_1, \ldots, p_n) \)

Representation of Points-to Sets

- \( \alpha :: \tau \times \lambda \)
  - \( \tau \) represents memory locations
  - \( \lambda \) represents procedures
- \( \tau :: \bot \mid \text{ref}(\alpha) \)
- \( \lambda :: \bot \mid \text{lam}(\alpha_1, \ldots, \alpha_n)(\alpha_{n+1}, \ldots, \alpha_{n+m}) \)
“Type-Rules”

- Analysis is expressed in “type rules”
- $\Delta \vdash x : \text{ref}(\alpha_1)$
- $\Delta \vdash y : \text{ref}(\alpha_2)$
- $\alpha_2 \leq \alpha_1$

$$\Delta \vdash \text{welltyped}(x=y)$$

More Rules

- $\Delta \vdash x : \text{ref}(\tau \times _)$
- $\Delta \vdash y : \tau$

$$\Delta \vdash \text{welltyped}(x=&y)$$

- $\Delta \vdash x : \text{ref}(\text{ref}(\alpha_1) \times _)$
- $\Delta \vdash y : \text{ref}(\alpha_2)$
- $\alpha_2 \leq \alpha_1$

$$\Delta \vdash \text{welltyped}(\ast x=y)$$
Example

```plaintext
a = &x
b = &y
if p then
  y = &z;
else
  y = &x;
fi
X = &y
```

Location Sets

- Use equivalence classes to represent “types”
  - Fast union / join using union-join data structure
  - \( \text{ecr}(x) \) is equivalence class for \( x \)
  - \( \text{Type}(\text{ecr}(x)) \) gives representative element of equivalence class
Inference Rules

- $x = y$
  
  let \( \text{ref}(\tau_1 x \lambda_1) = \text{type}(\text{ecr}(x)) \)
  
  \( \text{ref}(\tau_2 x \lambda_2) = \text{type}(\text{ecr}(y)) \) in
  
  if \( \tau_1 \neq \tau_2 \) then \( \text{cjoin}(\tau_1, \tau_2) \)
  
  if \( \lambda_1 \neq \lambda_2 \) then \( \text{cjoin}(\lambda_1, \lambda_2) \)

  \( \text{cjoin} \) conditionally joins the equivalence classes once one becomes $\neq \bot$

Rule for $*x = y$

- $*x = y$
  
  let \( \text{ref}(\tau_1 x \_ \lambda_1) = \text{type}(\text{ecr}(x)) \)
  
  \( \text{ref}(\tau_2 x \lambda_2) = \text{type}(\text{ecr}(y)) \) in
  
  if \( \tau_1 = \bot \) then \( \text{settype}(\tau_1, \text{ref}(\tau_2 x \lambda_2)) \)
  
  else let \( \text{ref}(\tau_3 x \lambda_3) = \text{type}(\tau_1) \) in
  
  if \( \tau_2 \neq \tau_3 \) then \( \text{cjoin}(\tau_3, \tau_2) \)
  
  if \( \lambda_2 \neq \lambda_3 \) then \( \text{cjoin}(\lambda_3, \lambda_2) \)