Dependence & Alias Analysis

CS2210
Lecture 17

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
  - \( S_1 \triangleright S_2 \)
- True (or flow)
  - \( S_1 \triangleright S_2 \)
- Anti-dependence
  - \( S_1 \triangleleft S_2 \)
- Output dependence
  - \( S_1 \triangleright S_2 \)
- Input “dependence”
  - \( S_1 \triangleright S_2 \) - two statements (instructions) read the same input
**Dependence DAG**

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

**Example:**
1. \( r3 := [r15](4) \)
2. \( r4 := [r15+4](4) \)
3. \( r2 := r3 - r4 \)
4. \( s5 := [s12](4) \)
5. \( s12 := r12 + 4 \)
6. \( s6 := s1 + s5 \)
7. \( [s15+4](4) := s3 \)
8. \( s5 := s6 + 2 \)

**Instruction Schedules**

- To find feasible schedule need to know latencies
- \( \text{Latency}(I_1, n_1, I_2, n_2) := \) number cycles latency incurred by beginning execution of \( I_2 \)'s \( n_2 \)th cycle while executing cycle \( n_1 \) of \( I_1 \)
- \( \text{Conflict}(I_1, I_2) = \) 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: $S_1 \cdot S_k$ extended for array statements:
  $S_1[i_1, \ldots, i_k] \cdot S_k[i_1', \ldots, i_k']$ asds
  - This means that either
    - $S_1$ precedes $S_k$ (in loop body) and
    - $<i_1, \ldots, i_k> < <i_1', \ldots, i_k'>$ OR
    - $S_1$ follows (or is equal to) $S_k$ and
    - $<i_1, \ldots, i_k> < <i_1', \ldots, i_k'>$

Example

```plaintext
for i1 := 1 to 3 do
  for i2 := 1 to 4 do
    S1: t := x + y
    S2: a[i1,i2] := b[i1,i2] + c[i1,i2]
    S3: b[i1,i2] := a[i1,i2-1] * d[i1+1,i2]
  endfor
endfor
```

- Sequences
  - $S2[i1,2-1] \cdot S3[i1,2]$
  - $S2[i1,2] \cdot S3[i1,2]$
- Dependences
  - $S2[i1,2-1] \cdot S3[i1,2]$
  - $S2[i1,2] \cdot S3[i1,2]$

Vectors

- **Distance Vector:**
  - $d = <d_1, \ldots, d_k>$ means that iteration
    $<i_1 + d_1, \ldots, i_k + d_k>$ depends on iteration
    $<i_1, \ldots, i_k>$
- **Direction Vector:**
  - $d = <d_1, \ldots, d_k>$ 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 $d_i^\pm \in \mathbb{Z} \cup \{ -\infty, \infty \}$
- Loop-independent and loop-carried dependences
- $d_i^-$ independent
- $d_i^+$ carried by loop at nesting level $i$
- $S_2[i_1,i_2-1] b_i S_3[i_1,i_2]$
- $S_2[i_1,i_2] \delta_0 a S_3[i_1,i_2]$
- Distance vector notation:
  - $S_2[i_1,i_2-1] < 0,1>$
  - $S_3[i_1,i_2] < 0,0>$
- $S_2[i_1,i_2] < 0,0>$
- $S_3[i_1,i_2]$

Dependence Testing
- Determine dependences between loop iterations
  - E.g., to reorder statements, parallelize
  - Diophantine equations:
    - $2i_1 + 1 = 3i_2 - 5$
    - Constrained: $1 \leq i_1, i_2 \leq 4$
    - Solution in general NP
  - So only sufficient conditions for non-dependence

Dependence Test
- for $i_1 := 1$ to $h_1$
  - for $i_2 := 1$ to $h_2$
    - for $i_n := 1$ to $h_n$
      - $x_{i_1} \cdots a_i x_{i_n} _{i_1} \cdots b_i x_{i_n}$
      - $\cdots$ endfor
    - $\cdots$ endfor
- GCD test:
  - If
    - $\gcd(a_{i_1}, b_{i_2}) | s_{i_1,i_2}(R-5)$
      - then references to $x$ are independent
    - $\gcd(a_{i_1}, b_{i_2}) = \pm 1$ if direction of $j$
      - $S = \text{else } x, b_i$
    - For $a_{j=1} \cdots b_{j=1}$:
      - $\gcd(2-2) \mid (-5 - 1 + 3 - 2)$ may have same iteration dependence
      - $\gcd(3,2) \mid -5$ may have inter-iteration dependence
GCD Test Example

- \( \text{gcd}(3-2) | (-5 \cdot 1 + 3 - 2) \) may have same iteration dependence.
- \( \text{gcd}(3,2) | -5 \) may have inter-iteration dependence.
- But not \( \text{gcd}(4-2) | (-1+4-2) \)
- And not \( \text{gcd}(4,2) | 1 \)

For \( i := 1 \) to 4 do
   \( b[j] = a[3 \cdot i-5]+2.0 \)
   \( a[2 \cdot i+1] := 1.0/i \)
Endfor

for i := 1 to 4 do
   \( b[j] = a[4 \cdot i]+2.0 \)
   \( a[2 \cdot i+1] := 1.0/i \)
Endfor

Separability

- Two array references are separable := in each pair of subscript positions the expressions are of the form \( a^n i + b_1 \) and \( a^n j + b_2 \)
  - Weakly separable := \( a^n i + b_1 \) and \( a^n j + b_2 \)
- For separable references dependence testing is trivial, a dependence exist if
  - \( a=0 \) and \( b_1=b_2 \) or \( b_1, b_2 \) \( | a <= n \)
  - 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:
  
  ```text
  ?
  ```

**Memory Disambiguation**

- Want to disambiguate memory accesses for optimization:

```c
exam1() {
  int a,x;
  extern int* q;
  ...
  k = a+5;
  f(a,&k);
  *q = 3;
  k = a+5; /* is this redundant */
  ...
}
```

**Alias Information Classification**

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

```text
```
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 = *y
- x = op(y₁, ... yₙ)
- x = allocate(y)
- *x = y
- x = fun(f₁...fₙ)→(r₁...rₘ)S'
- x₁, ..., xₙ = p(y₁, ..., pₙ)

Representation of Points-to Sets
- α :: τ X λ
  - τ represents memory locations
  - λ represents procedures
- τ :: ⊥ | ref(α)
- λ :: ⊥ | lam(α₁ ... αₙ)(α₁₊₁ ... αₙ₊ₙ)
"Type-Rules"

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

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

More Rules

- $A \vdash x : \text{ref}(\tau) \quad A \vdash y : \tau$

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

- $A \vdash x : \text{ref}(\text{ref}(\alpha_1) \ x) \quad A \vdash y : \text{ref}(\alpha_2)$
- $\alpha_2 \leq \alpha_1$

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

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
- $ecr(x)$ is equivalence class for $x$
- $Type(ecr(x))$ gives representative element of equivalence class

Inference Rules

- $x = y$
  let $ref(\tau_1 \times \lambda_1) = type(ecr(x))$
  $ref(\tau_2 \times \lambda_2) = type(ecr(y))$ in
  if $\tau_1 \neq \tau_2$ then $cjoin(\tau_1, \tau_2)$
  if $\lambda_1 \neq \lambda_2$ then $cjoin(\lambda_2, \lambda_2)$
  $cjoin$ conditionally joins the equivalence classes once one becomes $\neq \bot$

Rule for *$x = y$

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