Dependence & Alias Analysis

CS2210
Lecture 18

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)

CS2210 Compiler Design 2004/5

Array Variable Dependences

- Dependences are a function of
  - Index variables
  - Statements
- Statement sequence: $S_1 \rightarrow S_k$ extended for array statements:
  - $S_i[i_1, \ldots, i_k] \rightarrow S_j[i_2, \ldots, i_k]$
  - This means that either
    - $S_i$ precedes $S_j$ (in loop body) and $<i_1, \ldots, i_k> < <i_2, \ldots, i_k>$ OR
    - $S_i$ follows (or is equal to) $S_j$ and $<i_1, \ldots, i_k> < <i_2, \ldots, i_k>$

Example

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

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:**
  - $d$ is an 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

- \( d = \langle [d_1^-, d_1^+], ..., [d_k^-, d_k^+] \rangle \) where \( d \in \mathbb{Z} \cup \{-\infty, \infty\} \)
- Loop-independent and loop-carried dependences
- \( d_0 \) independent
- \( d_i \) carried by loop at nesting level \( i \)
- \( S_2[i_1,i_2-1] \delta S_3[i_1,i_2] \)
- \( S_2[i_1,i_2] \delta_0 a S_3[i_1,i_2] \)
- \( S_2[i_1,i_2] \delta_0 a S_3[i_1,i_2] \)
- Distance vector notation:
  - \( S_2[i_1,i_2-1] <0,1> \)
  - \( S_3[i_1,i_2] <0,0> \)

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 \( h_n \)
    - \( x[\ldots, a_j^+, a_j^-, b_j^+, b_j^-, \ldots] = \ldots x[\ldots, b_j^+, b_j^-, a_j^+, a_j^-, \ldots] \)
  - endfor
- endfor
- endfor

GCD test:
- If
  - NOT gcd(b_{i_1,j}, sep(a_{i_1,j},b_{ i_1,j}), j) \mid \sum_{j=0..n} (a_{i_1,j} - b_{i_1,j})
    then references to \( x \) are independent
  - sep(a_{i_1,j},b_{i_1,j}) = \{a-b\} if direction \( j \) is even \( (a,b) \)
    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 -1 + 3 -2) may have inter-iteration dependence
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.

Separability

- Two array references are separable :=
  in each pair of subscript positions the expressions are of the form $a^n_j + b_1$ and $a^n_j + b_2$.
  - Weakly separable := $a^n_j + b_1$ and $a^n_j + b_2$
- For separable references, dependence testing is trivial, a dependence exist if
  - $a = 0$ and $b_1 = b_2$
  - $(b_1 - b_2) / a = n_j$
- 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:
  - \( p = &x; \) \( x \) 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></td>
<td></td>
</tr>
<tr>
<td>Take control flow into account</td>
<td>If (d) ( p = &amp;x; ) { (x, *p) } else ( p = &amp;y; ) { (y, *p) } | (x, *p) }</td>
<td>If (d) ( p = &amp;x; ) { (x, *p) } else ( p = &amp;y; ) { (y, *p) } | (x, *p) }</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Flow-insensitive:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ignore control flow</td>
<td>If (d) ( p = &amp;x; ) { (x, *p) } else ( p = &amp;y; ) { (x, *p) } | (x, *p) }</td>
<td>const int* p = &amp;x; { (x, *p) } no other assignments to p throughout program</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Steensgaard’s Algorithm

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

Alias-relevant Statements

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

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

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

More Rules
- $A \vdash x: \text{ref}(\tau x\_)$
  - $A \vdash y: \tau$

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

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

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

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

Rule for $*x = y$

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

- Commonly used, e.g., in ORC
- One-level Flow (by Das) a recent improvement
  - use directional assignment
  - precision as good as Anderson’s algorithm but as fast as Steensgaard in practice

Early Compiler Optimizations

- early = close to frontend
  - algebraic optimizations, CSE, value numbering & others

Catalog of Early Optimizations

- Constant folding and propagation
  - Covered earlier
  - New: sparse conditional copy propagation -- requires SSA form
- Scalar replacement of aggregates
  - To enable register allocation of record fields
- Algebraic simplification and reassociation
- Value numbering
  - Discover expressions that compute identical values
  - At basic block level & globally
- Copy propagation
Scalar Replacement of Aggregates

- Simple, yet effective
- Given a record r
  - Identify which fields f have scalar values
  - Ensure that neither r nor f are aliased
  - Assign field to temporary of same type
  - Replace accesses to field by accesses to temporary

Algebraic Simplification & Reassociation

- Use algebraic properties of operators to simplify expressions
  - Reassociation = use commutativity / associativity / distributivity to simplify
- Example: for integers i
  - i+0 = 0+i = i
  - 0-i = -i
  - i*1 = 1*i = i
  - 0*i = 0

Algebraic Simplification

- Strength reduction:
  - i*2 = i + i
  - Addition is often faster than multiplication
    - On the Alpha integer multiply 8 cycles, addition 1 cycle!
    - i*5 = i<<2 + 1 (= 2 cycles versus 8)
- Caveats
  - (i+j) + (i-j) + (i+j) = 4*i-4*j
    - RHS overflows on 32-bit machine when i = 2^30 and j = 2^30-1
  - Exceptions: x = 1/0 (what if code is not reached?)
Addressing Expressions

- Overflow makes no difference
  - Unsigned operations so results guaranteed to be identical
  - All mathematical equalities are guaranteed to produce correct results
- **Canonicalization**
  - Optimization strategy:
    - Turn expression into sum of products form
    - Use commutativity to collect constant and loop-invariant parts together

Canonicalization Example

- For Pascal (row major order)
  
  ```pascal
  var a: array[lo1..hi1,lo2..hi2] of etype;
  do j = lo2 to hi2 begin a[i,j] := b + a[i,j] end
  Address(a[i,j]) = base_a + (i-lo1) * (hi2-lo2+1) +
  (hi1 * (hi2-lo2+1) -lo2) *w
  = (lo1 * (hi2-lo2+1) -lo2) *w + base_a + (hi2-
  lo2+1) *w + (lo2-1) *w
  Only have to calculate j*w in loop
  - Can strength-reduce to addition of w
  ```

Algebraic Simplification of Floating Point Expressions

- **Can almost never do it!** In particular the following are NOT true in FP arithmetic
  
  - x * 0 = 0
    - IEEE standard has representation for x = any NaN = NaN
    - NaN = Not a Number
    - Signaling and non-signaling NaNs (former cause exceptions, latter don’t)
  - x / 0 = x
    - If x is signaling NaN exception is raised when ‘/’ is executed
  - (a+b) +c = (a+b) +c
    - 1.0 + (MF-MF) = 1.0
    - (1.0 +MF) - MF = 0.0
Value Numbering

- Goal: identify expressions that have same value
- Approach: hash expression to a hash code
  - Then have to compare only expressions that hash to same value

Example

\[
\begin{align*}
  a &:= x \mid y \\
  b &:= x \mid y \\
  t1 &:= !z \\
  &\text{if } t1 \text{ goto } L1 \\
  x &:= !z \\
  c &:= x \& y \\
  t2 &:= x \& y \\
  &\text{if } t2 \text{ trap } 30
\end{align*}
\]

Global Value Numbering

- Generalization of value numbering for procedures
- Requires code to be in SSA form
- Crucial notion: congruence
  - Two variables are congruent if their defining computations have identical operators and congruent operands
    - Only precise in SSA form, otherwise the definition may not dominate the use
Value Graphs

- Labeled, directed graph
- Nodes are labeled with
  - Operators
  - Function symbols
  - Constants
- Edges point from operator (or function) to its operands
- Number labels indicate operand position

Example

```
entry
receive n(val)
  i_1 := 1
  j_1 := 1
  i_2 := phi(i_1, i_2)
  j_2 := phi(j_1, j_2)
  i_3 := i_2 + 1
  j_3 := j_2 + 1
B1

B2

B3

B4

B5

exit
```

Congruence Definition

- Maximal relation on the value graph
- Two nodes are congruent iff
  - They are the same node, or
  - Their labels are constants and the constants are the same, or
  - They have the same operators and their operands are congruent
- Variable equivalence := x and y are equivalent at program point p iff they are congruent and their defining assignments dominate p
Global Value Numbering Algo.

- Fixed point algorithm
- Partition nodes into congruent sets
- Initial partition: nodes with same label are in same partition
- Iterate: split partitions where operands are not congruent