CSC2/455 Software Analysis and Improvement
Dependence Testing

Sreepathi Pai

URCS

April 1, 2019

Outline

Outline

Loop optimizations so far

- Important applications
- Loop Dependences
- Identifying loop dependences
- Vectorization

Outline

The need for dependence testing

- Recall vectorization needs a dependence graph
  - If no dependence, can be vectorized!
  - Otherwise, need to figure out what the type of dependences exist
  - And their direction
- Other loop transformations (not just vectorization) will depend heavily on accurate dependence information
- General problem requires solving an ILP
  - Can we avoid this?

Recall: Dependence using Iteration Vectors

Let α and β be iteration vectors:

- $\alpha = (i_1, i_2, i_3, ..., i_N)$
- $\beta = (i'_1, i'_2, i'_3, ..., i'_N)$

Then a dependence exists if:

- (vectors) $\alpha < \beta$
- $fX(\alpha) = gX(\beta)$, for $1 <= X <= M$
- remember, fX and gX are assumed to be affine expressions
  - if they are not?

"Independence" Testing

- If no solutions to the ILP exist, then there are no dependences!
- Otherwise:
  - What is direction of dependence?
  - What level of the loop carries the dependence?

Exact and Conservative Tests

- Conservative test
  - Always correct when it determines there is no dependence
  - May be wrong when it determines there is a dependence
- Exact tests
  - Detects dependence if and only if they actually exist

Subscripts

```
DO i
  DO j
    DO k
      A(i, j) = A(i, k) + C
 ...
```

- First subscript: i and i
- Second subscript: j and k

ZIV, SIV and MIV

```
DO i
  DO j
    DO k
      A(5, i+1, j) = A(N, i, k) + C
    ENDDO
  ENDDO
ENDDO
```

- First subscript 5 and N are *zero-index variable* (ZIV) subscripts
- Second subscript $i + 1$ and i are *single-index variable* (SIV) subscripts
- Third subscript j and k are *multiple-index variable* (MIV) subscripts

Why ZIV, SIV and MIV matter

- Classifying subscripts allows tests tailored to each class of subscript
  - ZIV can simply be tested for equality
  - SIV tests are usually simpler than MIV tests
- Other notions that help simplify dependence testing:
  - Separability: a subscript does not share its index variables with other subscripts
  - Coupled: some subscripts share index variables
    `A(i+1,j,k-1)=A(i,j+i,k-1)`
  - See AK, Section 3.2 for more

The GCD test for integral solutions

The (linear) Diophantine equation:

$$a_1 i_1 + a_2 i_2 + ... + a_n i_n = c$$

has solutions only if the greatest common divisor (gcd):

$$gcd(a_1, a_2, ..., a_n)$$

divides c

Examples

- $2i - 2j = 1$
- $9x + 15y + 21z = 30$
- Consider these together:
  - $x - 2y + z = 0$
  - $3x + 2y + z = 5$
  - After solving for z?

GCD test

- The GCD test returns false for an affine equality if the gcd does not divide c
- So, for each affine expression:
  - Check if the GCD test returns false, this implies no dependence
- If the GCD test returns true for all equations:
  - Solve them using (e.g.) gaussian elimination
  - Redo the GCD test, if result is false implies no dependence
- GCD is not an exact test
  - Solutions may exist outside loop iteration space

Other heuristics

- Dragon Book 11.6.4
  - Independent Variables Test
  - Acyclic Test
  - The Loop Residue Test
- AK, Chapter 3
  - Various SIV tests
  - Banerjee's Inequality
  - Delta test
  - See 3.4.2 for more, esp. the Omega test

Outline

Operating on intermediate forms

- All examples so far on FORTRAN source code
  - with loops clearly marked
  - with loop indices and array indices easily related
- Can we operate on an intermediate representation?
  - Identify loops in CFGs
  - Identify loop *induction variables*

Identifying loops

- Back edge
  - An edge between nodes t and h, i.e. $t \rightarrow h$, where h dominates t
- A *natural* loop has a single-entry *header* node:
  - All nodes in the loop are dominated by the header
  - There is a *back edge* to the header node
- More formally, the natural loop for a back edge $n \rightarrow d$,
  - is the set of all nodes that can reach n without going through d
- Can be constructed using a depth-first search on the reverse CFG
  - See Algorithm 9.46 in the Dragon Book, Chapter 9, Section 9.6.6

Induction variables

An induction variable is a variable that changes by a fixed constant value every iteration of the loop

- Can be computed using a single addition/subtraction every iteration
- Can be used to eliminate multiplications
  - Strength reduction

The problem with induction variables

```
for(i = 1; i < 10; i++) {
   k = 3 * i;
   A[k] = A[k - 1] + A[k + 1];
}
```

- `k` is not an index variable
  - Can't use in dependence tests
- It is, however, an induction variable
  - Also, an affine function of the index variable

Substituting induction variables

```
for(i = 1; i < 10; i++) {
   A[3*i] = A[3*i - 1] + A[3*i + 1];
}
```

See AK, Chapter 4, Section 4.5, for algorithms.

Loop normalization

```
for(i = 3; i < 30; i+=3) {
   A[i] = A[i - 1] + A[i + 1];
}
```

After normalization:

```
for(i = 1; i < 10; i+=1) {
   A[3*i] = A[3*i - 1] + A[3*i + 1];
}
```

Loop Transformation Workflow

- Identify loops
- Identify induction variables
- Normalize loops
  - Loop bounds start from 0 (or 1)
  - Loop bounds increase by 1
  - All array index expressions only involve index variables or loop-invariant expressions
- Perform dependence analysis
- Transform code
  - (already seen) Vectorization
  - Others: today and next lecture

Loop Source Transformations

See Dragon Book, Section 11.7.8 for affine transformations that lead to source code transformations demo-ed on the remaining slides.

Loop Fusion

Before:

```
for(i = 1; i <= N; i++)
   Y[i] = Z[i];

for(j = 1; j <=N; j++)
   X[j] = Y[j];
```

After:

```
for(p = 1; p <= N; p++)
   Y[p] = Z[p];
   X[p] = Y[p];
```

Loop Fission

Before:

```
for(p = 1; p <= N; p++)
   Y[p] = Z[p];
   X[p] = Y[p];
```

After:

```
for(i = 1; i <= N; i++)
   Y[i] = Z[i];

for(j = 1; j <=N; j++)
   X[j] = Y[j];
```

Loop Re-indexing

Before:

```
for(i = 1; i <= N; i++)
   Y[i] = Z[i];
   X[i] = Y[i - 1];
```

After:

```
if(N >=1) X[1] = Y[0]
for(p = 1; p <= N-1; p++)  {
   Y[p] = Z[p];
   X[p+1] = Y[p];
}
if(N>=1) Y[N] = Z[N];
```

Loop Scaling

Before:

```
for(i = 1; i <= N; i++)
   Y[2*i] = Z[2*i];

for(j = 1; j <= 2*N; j++)
   X[j] = Y[j];
```

After:

```
for(p = 1; p <= 2*N; p++)  {
  if(p mod 2 == 0)
     Y[p] = Z[p];
   X[p] = Y[p];
}
```

Loop Reversal

Before:

```
for(i = 0; i <= N; i++)
   Y[N-i] = Z[i];

for(j = 0; j <= N; j++)
   X[j] = Y[j];
```

After:

```
for(p = 0; p <= N; p++)  {
   Y[p] = Z[N-p];
   X[p] = Y[p];
}
```

Loop Permutation

Before:

```
for(i = 1; i <=N; i++)
   for(j = 0; j <=M; j++)
       Z[i, j] = Z[i-1, j];
```

After:

```
for(p = 0; p <= M; p++)
   for(q = 0; q <= N; q++)
       Z[q, p] = Z[q-1, p];
```

Loop Skewing

Before:

```
for(i = 1; i < N + M - 1; i++)
   for(j = max(1, i+N); j <= min(i, M); j++)
       Z[i, j] = Z[i-1, j-1];
```

After:

```
for(p = 1; p<=N; p++)
   for(q = 1; q<=M; q++)
       Z[p, q-p] = Z[p-1, q-p-1];
```

Outline

References

- Allen and Kennedy, Chapter 3
- Alternatively, Chapter 11, Sections 11.4 and 11.6 of the Dragon book
- Section 11.7.8 of the Dragon Book