TU Wien

Institute of Information Systems Engineering
Research Division of Compilers and Languages

Übersetzer für Parallele Systeme
Compilers for Parallel Systems

LVA 185.A64

SS 2018

Hans Moritsch

hans.moritsch@tuwien.ac.at

Dependence and Transformations

- Correctness of dependence-based transformations
- When is it safe to apply program transformations?
  - transformed program is "equivalent" to original
  - if both produce the same sequence of states (all variables): too expensive and not necessary
- Two computations are equivalent if, on the same inputs
  - they produce identical values for output variables (at the time output statements are executed), and
  - the output statements are executed in the same order
  - $\Rightarrow$ they produce identical results
  - Elimination of exceptions occuring in the original program
- A transformation is *safe* if it results in an equivalent program

Reordering Transformations

- *Reordering transformation*
  - any program transformation that merely changes the order of execution of the code,
  - without adding or deleting any executions of any statements
  - it does not eliminate dependences
  - it may reverse it: interchange source and sink: incorrect
- A reordering transformation *preserves* a dependence, if it preserves the relative execution order of the source and sink of the dependence

Fundamental Theorem of Dependence:

- Any reordering transformation that preserves every dependence in a program preserves the meaning of that program.
- $\Rightarrow$ Original and transformed programs are equivalent
- A transformation is *valid* for the program to which it applies, if it preserves all dependences in the program
  - Note: this condition is stronger than equivalence

Example

```
L1   for i=1 to n

         update a[i,1] and a[i,2] :
L2       for j=1 to 2
S1           a[i,j] = a[i,j] + b
         endfor

         swap a[i,1] and a[i,2] :
S2       t = a[i,1]
S3       a[i,1] = a[i,2]
S3       a[i,2] = t

     endfor
```

$S_1 \delta S_2$
$S_1 \delta S_3$
$S_1 \delta S_4$

* interchange of update and swap is invalid
* but transformed program is equivalent

Direction Vector Example

```
L1  for i=1 to n
L2     for j=1 to m
L3        for k=1 to l
S            a[i,j,k-1] = a[i-1,j,k] + 10
          endfor
       endfor
    endfor
```

$$S \; \delta^{(<,=,>)}_{1,true} \; S$$

* Dependence is impossible, if

  leftmost non"="component $\neq$ "<"

* Level of loop carried dependence: position of leftmost non"="component of the direction vector of the dependence

Number of Dependences

```
L1  for i=1 to 10
L2     for j=1 to 99
S1          a[j,i] = b[j-1,i] + x
S2          c[j,i] = a[100-i,j] + y
       endfor
    endfor
```

* dependences exist between different pairs of statement instances
* consider different direction vector or different type only

Direction Vectors and Transformations

Direction Vector Transformation:

* A transformation
  * that is applied to a loop nest, and that
  * does not rearrange the statements in the body of the loop,
* is valid, if, after it is applied
  * none of the direction vectors for dependences with source and sink in the loop nest has a leftmost non-"=" component that is ">"
  * i.e., none of the dependences have been reversed
* e.g. (=,=,>,*) ... negative distance

Example: Loop Interchange

```
L1  for i=1 to 100
L2     for j=1 to 100
S         a[i,j] = a[i,j-1] * b[i,j]
       endfor
    endfor
```

$S\ \delta^{(=,<)}\ S$

```
L1  for j=1 to 100
L2     for i=1 to 100
S         a[i,j] = a[i,j-1] * b[i,j]
       endfor
    endfor
```

$S\ \delta^{(<,=)}\ S$

Example: Loop Interchange

```
L1  for i=1 to 100
L2     for j=1 to 100
S         a[i,j] = a[i-1,j+1] * b[i,j]
       endfor
    endfor
```

$S\ \delta^{(<,>)}\ S$

```
L1  for j=1 to 100
L2     for i=1 to 100
S         a[i,j] = a[i-1,j+1] * b[i,j]
       endfor
    endfor
```

$dir[S,S] = (>,<)$

dependence eliminated $\Leftrightarrow$ loop interchange invalid

Reordering Transformations Loop Carried Dependence

- A dependence will be said to be *satisfied* if transformations that fail to preserve it are avoided
  - e.g. satisfy all loop carried dependences at a certain level

level-k Dependences:

- Any reordering transformation that
  - (1) preserves the iteration order of the level-k loop
  - (2) does not interchange any loop at level < k to a position inside the level-k loop
  - (3) does not interchange any loop at level > k to a position outside the level-k loop
  - preserves all level-k dependences.

- Running all loops outside of and including the level-k loop sequentially preserves the level-k dependences

Reordering Transformations Example

```
for i=1 to 10
 S1  a[i+1] = f[i]
 S2  f[i+1] = a[i]
endfor
```

```
for i=1 to 10
 S1  f[i+1] = a[i]
 S2  a[i+1] = f[i]
endfor
```

dependences at level 1, and iteration order of level-1 loop is preserved

```
for i=1 to 10
   for j=1 to 10
      for k=1 to 10
       S a[i+1,j+2,k+3] =
                a[i,j,k]+b
      endfor
   endfor
endfor
```

```
for i=1 to 10
   for k=10 to 1 step -1
     for j=1 to 10
       S a[i+1,j+2,k+3] =
                a[i,j,k]+b
    endfor
   endfor
endfor
```

Arbitrary transformations inside the deepest dependence level

only level 1 dependence, and iteration order of i-loop is preseved

Reordering Transformations
Loop Independent Dependence

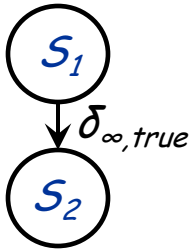
Loop-Independent Dependences

* If there is a loop-independent dependence from statement S_1 to statement S_2, any reordering transformation that
  * preserves the relative order of S_1 and S_2 in the loop body
  * and does not move statement *instances* between iterations,
* preserves that dependence.

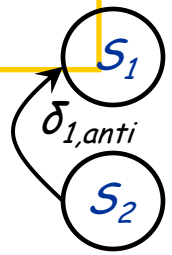
```
for i=1 to n
 S1  a[i] = b[i]+c
 S2  d[i] = a[i]+e
endfor
```

```
d[1] = a[1]+e
for i=2 to n
 S1  a[i-1] = b[i-1]+c
 S2  d[i] = a[i]+e
endfor
d[1] = a[1]+e
```

☹



order of statements is maintained, but *instances* are moved out of the loop



Reordering of Loop Iterations

Iteration Reordering

* A transformation that
  * reorders the iterations of a level-k loop
  * without making any other changes
* is valid
  * if the loop carries no dependence.

```
for i=1 to 10
 S1  a[i] = b[i]*2
 S2  c[i] = a[i]+3
endfor
```

```
for i=10 to 1 step -1
 S1  a[i] = b[i]*2
 S2  c[i] = a[i]+3
endfor
```

Parallelization – Basic Approach

- Create a thread for each iteration
- Run all threads asynchronously
  - $\Leftrightarrow$ reordering transformation (nondeterministic order)
- Statements from different iterations are executed concurrently

Loop Parallelization

- It is valid to convert a sequential loop to a parallel loop if the loop carries no dependence.
- interleaved execution of statements from different iterations must not reverse dependences
  - case 1: δ_∞ or δ_c in loop k or inner loop ($c \geq k$)
    - source and sink are in same iteration ... same thread, same sequence of execution $\Rightarrow$ preserved
  - case 2: δ_c in outer loop ($c < k$)
    - carried outside the k-loop $\Rightarrow$ preserved