Optimizing Loop Transformations

and data dependence analysis for loops

Christoph Kessler, IDA,
Linköpings universitet, 2009.

Why Loop Optimizations?

Loops are a promising object for compiler optimizations:

- High execution frequency
  - Most computation done in (inner) loops
  - Even small optimizations can have large impact
- Regular, repetitive behavior
  - compact description
  - *relatively* simple to analyze statically
- Well researched

Loop Optimizations – General Issues

- Move loop invariant computations out of loops
- Modify the order of iterations or parts thereof

Goals:

- Improve data access locality
- Faster execution
- Reduce loop control overhead
- Enhance possibilities for loop parallelization or vectorization

Only transformations that preserve the program semantics (its input/output behavior) are admissible

- Conservative (static) criterium: preserve data dependences
- Need data dependence analysis for loops
  (→ separate slide set)

Loop Invariant Code Hoisting

- **Move loop invariant code out of the loop:**
  - Example:

```
for (i=0;  i<10;  i++)
   a[i] = b[i]  + c / d;
```

```
tmp = c / d;
for (i=0;  i<10;  i++)
   a[i] = b[i]  + tmp;
```


Loop Unrolling

- **Loop unrolling**
  - Example:

```
for (i=0; i<50; i++) {
   a[i] = b[i];
}
```

```
for (i =0; i<50; i+=2) {
   a[i] = b[i];
   a[i+1] = b[i+1];
}
```

  - Reduces loop overhead
    (total # comparisons, branches, increments)
  - May enable further local optimizations
  - Drawback: longer code

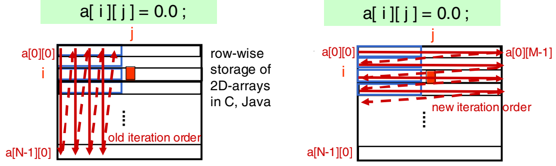
→ Exercise: Formulate the unrolling rule for statically unknown upper loop limit

Loop Interchange (1)

- For properly nested loops
  (statements in innermost loop body only)
  - Example 1:

```
for (j=0;  j<M;  j++)
  for (i=0;  i<N;  i++)
    a[ i ][ j ] = 0.0 ;
```

```
for (i=0;  i<N;  i++)
  for (j=0;  j<M;  j++)
    a[ i ][ j ] = 0.0 ;
```



  - Can improve data access locality
    (fewer cache misses / page faults)

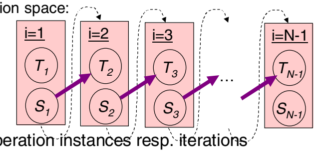
Foundations: Loop-Carried Data Dependences

```
S:  z = … ;
    …
T:  … = ..z.. ;
```

- Recall:
  Data dependence *S* → *T*,
  if operation *S* *may* execute (dynamically) before operation *T*
  and both *may* access the same memory location
  and at least one of these accesses is a write
  - In general, only a conservative over-estimation can be determined statically.
- Data dependence *S*→*T* is called ***loop carried*** by a loop *L*
  if the data dependence S→T may exist for instances of *S* and *T*
  in different iterations of *L*.
  - Example:

```
L:  for (i=1; i<N; i++) {
T_i:      …  =  x[ i-1 ];
S_i:      x[ i ] = …;
    }
```

Iteration space:



→ partial order between the operation instances resp. iterations

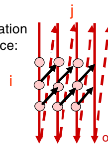
Loop Interchange (2)

- **Be careful** with loop carried data dependences!
  - Example 2:

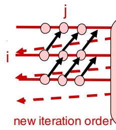
```
for (j=1;  j<M;  j++)
  for (i=0;  i<N;  i++)
    a[i][j] =…a[i+1][j-1]…;
```

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

Iteration space:



Iteration (*j*,*i*) reads location a[*i+1*][*j-1*] that was written in an earlier iteration, (*i-1*,*j+1*)



Iteration (*i*,*j*) reads location a[*i+1*][*j-1*], that will be over-written in a later iteration (*i+1*,*j-1*)

  - Interchanging the loop headers would violate the partial iteration order given by the data dependences

Loop Interchange (3)

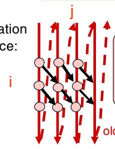
- **Be careful** with loop-carried data dependences!
  - Example 3:

```
for (j=1;  j<M;  j++)
  for (i=1;  i<N;  i++)
    a[i][j] =…a[i-1][j-1]…;
```

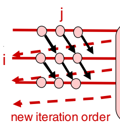
OK

```
for (i=1;  i<N;  i++)
  for (j=1;  j<M;  j++)
    a[i][j] =…a[i-1][j-1]…;
```

Iteration space:



Iteration (*j*,*i*) reads location a[*i-1*][*j-1*] that was written in earlier iteration (*j-1*,*i-1*)



Iteration (*i*,*j*) reads location a[*i-1*][*j-1*] that was written in earlier iteration (*i-1*,*j-1*)

  - Generally: Interchanging loop headers is only admissible if loop-carried dependences have the same direction for all loops in the loop nest (all directed along or all against the iteration order)

Loop Fusion

- Merge subsequent loops with same header
  - Example:

```
for (i=0;  i<N;  i++)
  a[ i ] = … ;
for (i=0;  i<N;  i++)
  … = … a[ i ] … ;
```

```
for (i= 0;  i<N;  i++) {
  a[ i ] = … ;
  … = … a[ i ] … ;
}
```

OK – Read of a[*i*] still after write of a[*i*], for all *i*

  - Can improve data access locality and reduce number of branches

Data Dependence Analysis

A more formal introduction

Christoph Kessler, IDA,
Linköpings universitet, 2009.

Data Dependence Analysis – Overview

- Important for loop optimizations, vectorization and parallelization, instruction scheduling, data cache optimizations
- Conservative approximations to disjointness of pairs of memory accesses
  - weaker than data-flow analysis
  - but generalizes nicely to the level of individual array element
- Loops, loop nests
  - Iteration space
  - Array subscripts in loops
  - Index space
- Dependence testing methods
- Data dependence graph
- Data + control dependence graph
  - Program dependence graph

Precedence relation between statements

S_1 statically (textually) precedes S_2 $\quad$ S_1 pred S_2

S_1 dynamically precedes S_2 $\quad$ $S_1 \lhd S_2$

S_1: $s \leftarrow 0$
for i **from** 1 **to** n **do**
S_2: $s \leftarrow s + a[i]$
S_3: $a[i] \leftarrow s$
od

Control and Data Depencence; Dependence Graph

Dependence = constraint on execution order

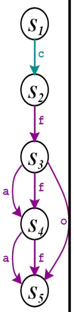
- control dependence by control flow: $S_1 \delta^c S_2$
- data dependence:

flow / true dependence: $S_3 \, \delta^f \, S_4$
$S_3 \lhd S_4$ and $\exists b$: S_3 writes b, S_4 reads b

anti-dependence: $S_3 \, \delta^a \, S_4$
$S_3 \lhd S_4$ and $\exists c$: S_3 reads c, S_4 writes c

output dependence: $S_3 \, \delta^o \, S_5$
$S_3 \lhd S_5$ and $\exists b$: S_3 writes b, S_5 writes b

S_1: **if** (e) **goto** S_3
S_2: $a \leftarrow ...$
S_3: $b \leftarrow a * c$
S_4: $c \leftarrow b * f$
S_5: $b \leftarrow x + f$



Data Dependence Graph

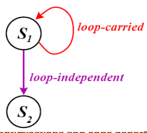
Data dependence graph: directed graph
(statements, precedence constraints due to data dependences)

For basic blocks:
acyclic dependence graph (DAG)

Within loops, loop nests: pred $\neq$ $\lhd$

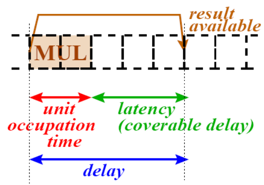
edge if dependence may exist for *some* pair of iterations
$\rightarrow$ cycles possible
loop-independent versus loop-carried dependences

```
  for (i=1; i<n; i++) {
S1:  a[i] = b[i] + a[i-1];
S2:  b[i] = a[i];
  }
```



Remark: Target-Level Dependence Graphs

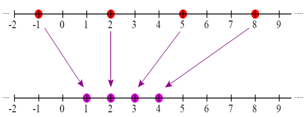
- For VLIR / target code, the dependence edges may be labeled with latency or delay of the operation
- See lecture on instruction scheduling for details



Loop Normalization

Given a loop of the form

for I **from** L **to** U **step** S **do**
... I ...
od



normalize the loop:

- lower bound 0 (C) resp. 1 (Fortran)
- step size +1

$\rightarrow$ update all occurrences of the loop counter I by $i*S-S+L$

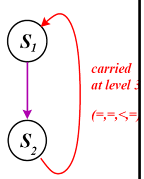
for i **from** 1 **to** $(U-L+S)/S$ **step** 1 **do**
... $(i*S-S+L)$ ...
od
$I \leftarrow i*S-S+L$

Loop Iteration Space

Beyond basic blocks: pred $\neq$ $\lhd$

Canonical loop nest: (HIR code)

for i_1 **from** 1 **to** n_1 **do**
for i_2 **from** 1 **to** n_2 **do**
$\ddots$
for i_k **from** 1 **to** n_k **do**
$S_1(i_1,...,i_k)$: $A[i_1, 2*i_3] \leftarrow B[i_2, i_3] + 1$
$S_2(i_1,...,i_k)$: $B[i_2, i_3 + i_4] \leftarrow 2 * A[i_1, 2*i_3]$



Iteration space: $ItS = [1..n_1] \times [1..n_2] \times ... \times [1..n_k]$

(the simplest case: rectangular, static loop bounds)

Iteration vector $\vec{i} = \langle i_1, ..., i_k \rangle \in ItS$

Dependence Distance and Direction

Lexicographic order on iteration vectors $\rightarrow$ dynamic execution order:

$S_1(\langle i_1,...,i_k\rangle) \lhd S_2(\langle j_1,...,j_k\rangle)$ iff
either S_1 pred S_2 and $\langle i_1,...,i_k\rangle) \leq_{lex} \langle j_1,...,j_k\rangle$
or $S_1 = S_2$ and $\langle i_1,...,i_k\rangle) <_{lex} \langle j_1,...,j_k\rangle$

distance vector $\vec{d} = \vec{j} - \vec{i} = \langle j_1 - i_1, ..., j_k - i_k\rangle$

direction vector $dirv = \text{sgn}(\vec{j} - \vec{i}) = \langle \text{sgn}(j_1 - i_1), ..., \text{sgn}(j_k - i_k)\rangle$
in terms of symbols = $< \; > \; \leq \; \geq$ *

Example: $S_1(\langle i_1,i_2,i_3,i_4\rangle) \; \delta^f \; S_2(\langle i_1,i_2,i_3,i_4\rangle)$
distance vector $\vec{d} = \langle 0,0,0,0\rangle$, direction vector $dirv = \langle =,=,=,=\rangle$,
loop-independent dependence

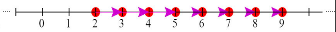
Example: $S_2(\langle i_1,i_2,i_3,i_4\rangle) \; \delta^f \; S_1(\langle i_1,i_2,i_3+i_4,i_4\rangle)$
distance vector $\vec{d} = \langle 0,0,?,0\rangle$, direction vector $dirv = \langle =,=,>,=\rangle$,
loop-carried dependence (carried by i_3-loop / at level 3)

Example

for i **from** 2 **to** 9 **do**
S_1 $X[i] \leftarrow Y[i] + Z[i]$
S_2 $A[i] \leftarrow X[i-1] + 1$
od

	$i=2$	$i=3$	$i=4$	...
S_1	$X[2] \leftarrow Y[2] + Z[2]$	$X[3] \leftarrow Y[3] + Z[3]$	$X[4] \leftarrow Y[4] + Z[4]$	...
S_2	$A[2] \leftarrow X[1] + 1$	$A[3] \leftarrow X[2] + 1$	$A[4] \leftarrow X[3] + 1$	...

There is a loop-caried, forward, flow dependence from S_1 to S_2.

Iteration space dependence graph:


Dependence Equation System

One-dimensional array A accessed in k nested loops: S_1 : $...A[f(\vec{i})]...$
S_2 : $...A[g(\vec{i})]...$

Is there a dependence between $S_1(\vec{i})$ and $S_2(\vec{j})$ for some $\vec{i}, \vec{j} \in ItS$?

typically f, g linear: $f(\vec{i}) = a_0 + \sum_{l=1}^{k} a_l i_l$, $g(\vec{i}) = b_0 + \sum_{l=1}^{k} b_l i_l$,

Exist $\vec{i}, \vec{j} \in \mathbb{Z}^k$ with $f(\vec{i}) = g(\vec{j})$, i.e., $a_0 + \sum_{l=1}^{k} a_l i_l = b_0 + \sum_{l=1}^{k} b_l j_l$, dep. equation

subject to $\vec{i}, \vec{j} \in ItS$, i.e.,

$1 \leq i_1 \leq n_1$, $1 \leq j_1 \leq n_1$,
$\vdots$ $\vdots$ iter. space constraints: linear inequalities
$1 \leq i_k \leq n_k$, $1 \leq j_k \leq n_k$

$\Rightarrow$ constrained linear Diophantine equation system $\rightarrow$ ILP (NP-complete)

Linear Diophantine Equations

$$\sum_{j=1}^{n} a_j x_j = c$$

where $n \geq 1$, $c,\; a_j \in \mathbb{Z}$, $\exists j:\; a_j \neq 0$, $x_i \in \mathbb{Z}$

Example 1: $x + 4y = 1$
has infinitely many solutions, e.g. $x = 5$ and $y = -1$.

Example 2: $5x - 10y = 2$
has no solution in $\mathbb{Z}$: absolute term must be multiple of

Theorem:
$\sum_{j=1}^{n} a_j x_j = c$ has a solution **iff** $\gcd(a_1, ..., a_b) | c$.

Proof: see e.g. [Zima/Chapman p. 143]

Dependence Testing, 1: GCD-Test

Often, a simple test is sufficient to prove independence: e.g.,

gcd-test [Banerjee'76], [Towle'76]:
independence if

$$\gcd\left(\bigcup_{l=1}^{n}\{a_l, b_l\}\right) \nmid \sum_{l=0}^{n}(a_l - b_l)$$

constraints on ItS not considered

Example: **for** i **from** 1 **to** 4 **do**
S_1 : $b[i] \leftarrow a[3*i-5]+2$
S_2 : $a[2*i+1] \leftarrow 1.0/i$

solution to $2i+1 = 3j-5$ exists in $\mathbb{Z}$ as $\gcd(3,2) | (-5-1+3-2)$
not checked whether such i, j exist in $\{1, ..., 4\}$

For multidimensional arrays?

subscript-wise test vs. linearized indexing

for i ...	**for** i ...	
S_1 : $...A[x[i], 2*i]...$	S_1 : $...A[i,i]...$	$A[i*(s_1+1)]$
S_2 : $...A[y[i], 2*i+1]...$	S_2 : $...A[i,i+1]...$	$A[i*(s_1+1)+1]$

Moreover:
Hierarchical structuring of dependence tests [Burke/Cytron'86]

Survey of Dependence Tests

gcd test

separability test (gcd test for special case, exact)

Banerjee-Wolfe test [Banerjee'88] rational solution in ItS

Delta-test [Goff/Kennedy/Tseng'91]

Power test [Wolfe/Tseng'91]

Simple Loop Residue test [Maydan/Hennessy/Lam'91]

Fourier-Motzkin Elimination [Maydan/Hennessy/Lam'91]

Omega test [Pugh/Wonnacott'92]

Loop Parallelization

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; i<n; i++)
     for (j=1; j<m; j++)
       for (k=1; k<r; k++)
S:       a[i][j][k] = ... a[i][j-1][k] ...          (=,<,=)
```

It is valid to convert a sequential loop to a parallel loop
if it does not carry a dependence.

```
   for (i=1; i<n; i++)        -->       forall ( i, 1, n, p )
S:   b[i] = 2 * c[i];                     b[i] = 2 * c[i];
```


Loop Transformations

Goal

+ modify execution order of loop iterations
+ preserve data dependence constraints

Motivation

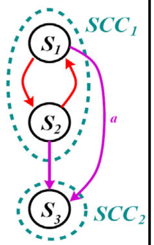
+ data locality
  → increase reuse of registers, cache
+ parallelism
  → eliminate loop-carried dependences, increase granularity
+ reduced overhead

Loop Distribution (a.k.a. Loop Fission)

```
    for (i=1; i<n; i++) {
S1:   a[i+1] = b[i-1] + c[i];
S2:   b[i]   = a[i] * k;
S3:   c[i]   = b[i] - 1;
    }
```

↓ Loop distribution

```
    for (i=1; i<n; i++) {
S1:   a[i+1] = b[i-1] + c[i];
S2:   b[i]   = a[i] * k;
    }
    for (i=1; i<n; i++)
S3:   c[i]   = b[i] - 1;
```



Safe if all statements forming a SCC in the dependence graph
end up in the same loop.
Forward (loop-carried) dep's are ok, but keep topological order.

+ often enables vectorization; better cache utilization of each loop.

Loop Fusion

for i **from** 1 to N **do**
  $c[i] \leftarrow a[i] + b[i]$
od
for j **from** 1 to N **do**
  $d[j] \leftarrow a[j] * e[j]$
od

fuse:

for i **from** 1 to N **do**
  $c[i] \leftarrow a[i] + b[i]$
  $d[i] \leftarrow a[i] * e[i]$
od

For array a large enough,
$a[i]$ will no longer be cached.

find second $a[i]$ in the cache
or even in a register

Safe if neither loop carries a (backward) dependence.

+ locality: can convert inter-loop reuse to intra-loop reuse
+ larger basic blocks
+ reduce loop overhead

Strip Mining / Loop Blocking / -Tiling

```
for (i=0; i<n; i++)
  a[i] = b[i] + c[i];
```

↓ loop blocking with block size s

```
for (i1=0; i1<n; i1+=s)              // loop over blocks
  for (i2=0; i2<min(n-i1,s); i2++)  // loop within blocks
    a[i1+i2] = b[i1+i2] + c[i1+i2];
```

Tiling = blocking in multiple dimensions + loop interchange

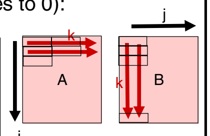
Goal: increase locality; support vectorization (vector registers)

Reverse transformation: Loop linearization

Tiled Matrix-Matrix Multiplication (1)

- Matrix-Matrix multiplication $C = A \times B$
  here for square ($n \times n$) matrices C, A, B, with n large (~10^3):
  - $C_{ij} = \Sigma_{k=1..n} A_{ik} B_{kj}$ for all $i, j = 1...n$
- Standard algorithm for Matrix-Matrix multiplication
  (here without the initialization of C-entries to 0):

```
for (i=0; i<n; i++)
  for (j=0;  j<n;  j++)
    for (k=0;  k<n;  k++)
      C[i][j] += A[i][k] * B[k][j];
```



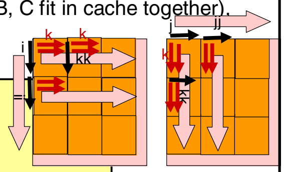
Good spatial locality on A, C

Bad spatial locality on B
(many capacity misses)

Tiled Matrix-Matrix Multiplication (2)

- Block each loop by block size S
  (choose S so that a block of A, B, C fit in cache together),
  then interchange loops
- Code after tiling:

```
for (ii=0; ii<n; ii+=S)
  for (jj=0;  jj<n;  jj+=S)
    for (kk=0;  kk<n;  kk+=S)
      for (i=ii;  i < ii+S;  i++)
        for (j=jj;  j < jj+S;  j++)
          for (k=kk;  k < kk+S;  k++)
            C[i][j] += A[i][k] * B[k][j];
```



Good spatial locality for A, B and C

Remark on Locality Transformations

- An alternative can be to change the data layout rather than the control structure of the program
  - **Example:** Store matrix B in transposed form,
    or, if necessary, consider transposing it, which may pay off over several subsequent computations
    - Finding the best layout for all multidimensional arrays is a NP-complete optimization problem
      [Mace, 1988]
  - **Example:** Recursive array layouts that preserve locality
    - Morton-order layout
    - Hierarchically tiled arrays

Loop Nest Flattening / Linearization

Flattens a multidimensional iteration space to a linear space:

for i **from** 0 **to** $n-1$ **do**
  for j **from** 0 **to** $m-1$ **do**
    iteration(i,j)
  od
od

linearize:

for k **from** 0 **to** $m \cdot n - 1$ **do**
  $i \leftarrow k \,/\, m$
  $j \leftarrow k \,\%\, m$
  iteration(i,j)
od

\+ larger iteration space, better for scheduling / load balancing

– overhead to reconstruct original iteration variables
may be reduced by using *induction variables* i, j
that are updated by accumulating additions instead of div and mod

Loop Unrolling

for i **from** 1 **to** 100 **do**
  $a[i] \leftarrow a[i] + b[i]$
od

unroll by 4:

for i **from** 1 **to** 100 **step** 4 **do**
  $a[i] \leftarrow a[i] + b[i]$
  $a[i+1] \leftarrow a[i+1] + b[i+1]$
  $a[i+2] \leftarrow a[i+2] + b[i+2]$
  $a[i+3] \leftarrow a[i+3] + b[i+3]$
od

\+ less overhead per useful operation

\+ longer basic blocks for local optimizations
(local CSE, local reg.-allocation, local scheduling, SW pipelining)

– longer code

Loop Unrolling with Unknown Upper Bound

for i **from** 1 **to** N **do**
  $a[i] \leftarrow a[i] + b[i]$
od

unroll by 4:

$i \leftarrow 1$
while $i+3 < N$ **do**
  $a[i] \leftarrow a[i] + b[i]$
  $a[i+1] \leftarrow a[i+1] + b[i+1]$
  $a[i+2] \leftarrow a[i+2] + b[i+2]$
  $a[i+3] \leftarrow a[i+3] + b[i+3]$
  $i \leftarrow i + 4$
od
while $i < N$ **do**
  $a[i] \leftarrow a[i] + b[i]$
  $i \leftarrow i + 1$
od

used e.g. in BLAS

Loop Unroll-And-Jam

unroll the outer loop
and fuse the resulting inner loops:

```
for i from 1 to N do
    for j from 1 to N do
        a[i] ← a[i] + b[j]
    od
od
```

unroll&jam:

```
for i from 1 to N step 2 do
    for j from 1 to N do
        a[i] ← a[i] + b[j]
        a[i+1] ← a[i+1] + b[j]
    od
od
```

The same conditions as for loop interchange (for the two innermost loops after the unrolling step) must hold (for a formal treatment see [Allen/Kennedy'02, Ch. 8.4.1]).

+ increases reuse in inner loop
+ less overhead

Loop Peeling

remove the first (or last) iteration of the loop
and clone the loop body for that iteration.

```
for i from 1 to N do
    a[i] ← (x+y)*b[i]
od
```

peel first iteration:

```
if N ≥ 1 then
    a[1] ← (x+y)*b[1]
    for i from 2 to N do
        a[i] ← (x+y)*b[i]
    od
fi
```

(Test on trip count can be removed if $N \geq 1$ is statically known.)

+ can enable loop fusion

+ may extract conditionals handling boundary cases from the loop

– longer code

Index Set Splitting

Divide the *iteration space* into two portions.

```
for i from 1 to 100 do
    a[i] ← b[i] + c[i]
    if i > 10 then
        d[i] ← a[i] + a[i−10]
    fi
od
```

split after 10:

```
for i from 1 to 10 do
    a[i] ← b[i] + c[i]
od
for i from 11 to 100 do
    a[i] ← b[i] + c[i]
    d[i] ← a[i] + a[i−10]
od
```

+ removes condition evaluation in every iteration
+ factors out the parallelizable set of iterations

– longer code

Loop Unswitching

```
for i from 1 to 100 do
    a[i] ← a[i] + b[i]
    if expression then
        d[i] ← 0
    fi
od
```

unswitch:

```
if expression then
    for i from 1 to 100 do
        a[i] ← a[i] + b[i]
        d[i] ← 0
    od
else
    for i from 1 to 100 do
        a[i] ← a[i] + b[i]
    od
fi
```

+ hoist loop-invariant control flow out of loop nest
+ no tests, no branches in loop body
  → larger basic blocks (see above), simpler software pipelining
– longer code

Scalar Replacement

For (inner) loops accumulating a value in an array element
use a temporary scalar for the accumulator variable:

```
for i from 1 to N do
    for j from 1 to N do
        a[i] ← a[i] + b[j]
    od
od
```

scalar repl.:

```
for i from 1 to N step 2 do
    t ← a[i]
    for j from 1 to N do
        t ← t + b[j]
    od
    a[i] ← t
od
```

+ keep t in a register all the time
+ saves many costly memory accesses to $a[i]$

Scalar Expansion / Array Privatization

promote a scalar temporary to an array to break a dependence cycle

```
for i from 1 to N do
    t ← a[i] + b[j]
    c[i] ← t+1
od
```

expand scalar t:

```
if N ≥ 1
    allocate t'[1..N]
    for i from 1 to N do
        t'[i] ← a[i] + b[j]
        c[i] ← t'[i]+1
    od
    t ← t'[N] // if t live on exit
fi
```

+ removes the loop-carried antidependence due to t
  → can now parallelize the loop!

- needs more array space

Loop must be countable, scalar must not have upward exposed uses.

May also be done conceptually only, to enable parallelization:
just create one private copy of t for every processor = array privatization

Outlook: Runtime Parallelization

Sometimes parallelizability cannot be decided statically.

if is_parallelizable(...)
  forall i **in** [0..n-1] **do** *// parallel version of the loop*
    iteration(i);
  od
else
  for i **from** 0 **to** $n-1$ **do** *// sequential version of the loop*
    iteration(i);
  od
fi

The runtime dependence test is_parallelizable(...) itself may partially run in parallel.

Outlook: Parallelization by Pattern Matching

Traditional loop parallelization fails for loop-carried dep. with distance 1:

```
S0:  s = 0;
     for (i=1; i<n; i++)
S1:     s = s + a[i];

S2:  a[0] = c[0];
     for (i=1; i<n; i++)
S3:     a[i] = a[i-1] * b[i] + c[i];
```

↓ Idiom recognition (pattern matching) [K.'94],...

```
S1': s = VSUM( a[1:n-1], 0 );

S3': a[0:n-1] = FOLR( b[1:n-1], c[0:n-1], mul, add );
```

↓ Algorithm replacement

```
S1'': s = par_sum( a, 0, n, 0 );
```


To be continued...

Christoph Kessler, IDA,
Linköpings universitet, 2009.