DF00100 Advanced Compiler Construction

TDDC86 Compiler optimizations and code generation

Optimization and Parallelization of Sequential Programs

Introduction to Data Dependence Analysis

Christoph Kessler

IDA / PELAB
Linköping University
Sweden

Outline

Towards (semi-)automatic parallelization of sequential programs

- Data dependence analysis for loops
  - Dependence tests
- Some loop transformations
  - Loop invariant code hoisting, loop unrolling, loop fusion, loop interchange, loop blocking and tiling, scalar expansion, and more
- Static loop parallelization
- Idiom recognition
- Run-time loop parallelization
  - Doacross parallelization
  - Inspector-executor method
  - If time permits: thread-level speculation

Foundations: Control and Data Dependence

- Consider statements *S, T* in a sequential program (*S*=*T* possible)
  - Scope of analysis is typically a function, i.e. intra-procedural analysis
  - Assume that a control flow path *S* … *T* is possible
  - Can be done at arbitrary granularity (instructions, operations, statements, compound statements, program regions)
  - Relevant are only the read and write effects on memory (i.e. on program variables) by each operation, and the effect on control flow

- **Control dependence** *S* → *T*, if the fact whether *T* is executed may depend on *S* (e.g. condition)
  - Implies that relative execution order *S* → *T* must be preserved when restructuring the program
  - Mostly obvious from nesting structure in well-structured programs, but more tricky in arbitrary branching code (e.g. assembler code)

Example:

```
S:  if (…) {
          …
T:        …
          …
      }
```

Foundations: Control and Data Dependence

- **Data dependence** *S* → *T*,
  if statement *S* *may* execute (dynamically) before *T* and both *may* access the same memory location and at least one of these accesses is a write
  - Means that execution order ”*S* before *T*” must be preserved when restructuring the program
  - In general, only a conservative over-estimation can be determined statically
  - **flow dependence**: (RAW, read-after-write)
    - *S* may write a location z that *T* may read
  - **anti dependence**: (WAR, write-after-read)
    - *S* may read a location x that *T* may overwrite
  - **output dependence**: (WAW, write-after-write)
    - both *S* and *T* may write the same location

Example:

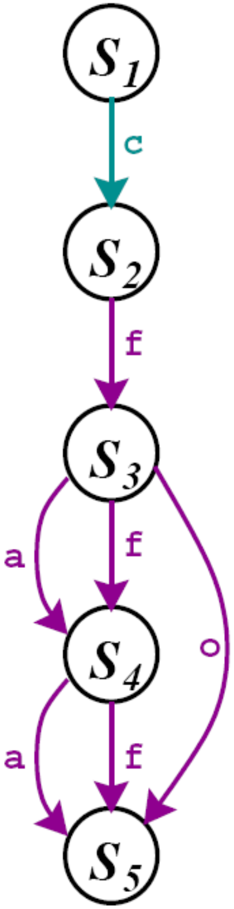
```
S:   z = … ;
     …
T:   … = ..z.. ;
```

(flow dependence)

Dependence Graph

- **(Data, Control, Program) Dependence Graph:** Directed graph, consisting of all statements as vertices and all (data, control, any) dependences as edges.

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



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

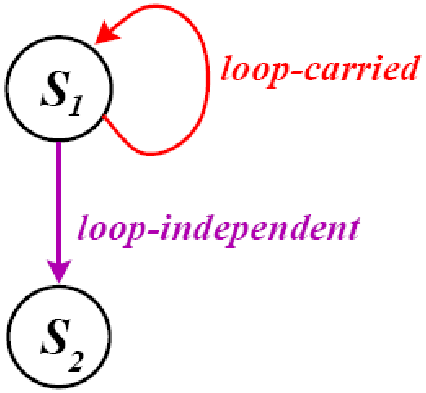
Data Dependence Graph

- **Data dependence graph for straight-line code** (”basic block”, no branching) is always acyclic, because relative execution order of statements is forward only.
- **Data dependence graph for a loop:**
  - Dependence edge *S*→*T* if a dependence may exist *for some pair of instances* (iterations) of *S, T*
  - Cycles possible
  - Loop-independent versus loop-carried dependences

Example:

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

(assuming that we know statically that arrays a and b do not intersect)



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
```

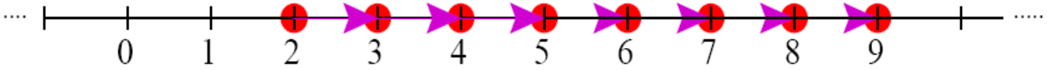
(assuming that we statically know that arrays A, X, Y, Z do not intersect, otherwise there might be further dependences)

	$i=2$	$i=3$	$i=4$	...
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	$A[2] \leftarrow X[1] + 1$	$A[3] \leftarrow X[2] + 1$	$A[4] \leftarrow X[3] + 1$	...

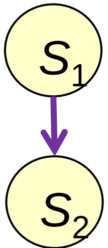
There is a loop-caried, forward, flow dependence from S_1 to S_2.

Iteration space dependence graph:

(Iterations unrolled)



Data dependence graph:



Why Loop Optimization and Parallelization?

Loops are a promising object for program optimizations, including automatic parallelization:

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

DF00100 Advanced Compiler Construction

TDDC86 Compiler optimizations and code generation

Data Dependence Analysis for Loops

A more formal introduction

Christoph Kessler, IDA,
Linköping University

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 S_1 pred S_2

S_1 dynamically precedes S_2 $S_1 \lhd S_2$

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

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

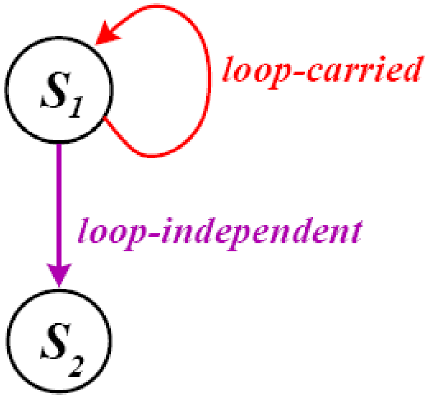
Data Dependence Graph

- **Data dependence graph for straight-line code** (”basic block”, no branching) is always *acyclic*, because relative execution order of statements is forward only.
- **Data dependence graph for a loop:**
  - Dependence edge *S*→*T* if a dependence *may* exist *for some pair of instances* (iterations) of *S, T*
  - Cycles possible
  - Loop-independent versus loop-carried dependences

Example:

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

(assuming we know statically that arrays a and b do not intersect)

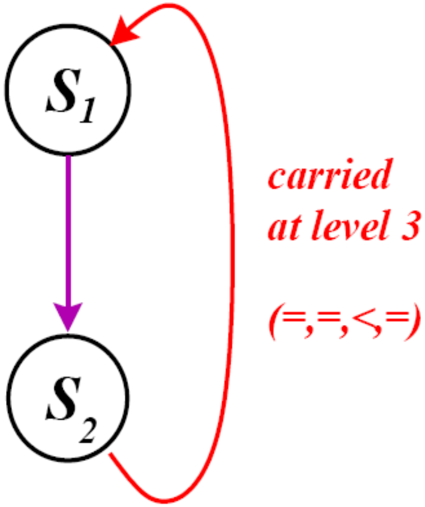


Loop Iteration Space

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
                $S_1(i_1, ..., i_k): \quad A[i_1, 2*i_3] \leftarrow B[i_2, i_3] + 1$
                $S_2(i_1, ..., i_k): \quad B[i_2, i_3 + i_4] \leftarrow 2 * A[i_1, 2*i_3]$
```



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

(the simplest case: rectangular, static loop bounds)

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

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
```

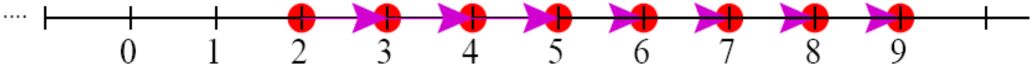
(assuming that we statically know that arrays A, X, Y, Z do not intersect, otherwise there might be further dependences)

	$i = 2$	$i = 3$	$i = 4$	...
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	$A[2] \leftarrow X[1] + 1$	$A[3] \leftarrow X[2] + 1$	$A[4] \leftarrow X[3] + 1$	...

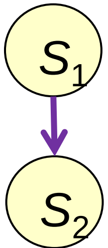
There is a loop-caried, forward, flow dependence from S_1 to S_2.

Iteration space dependence graph:

(Iterations unrolled)



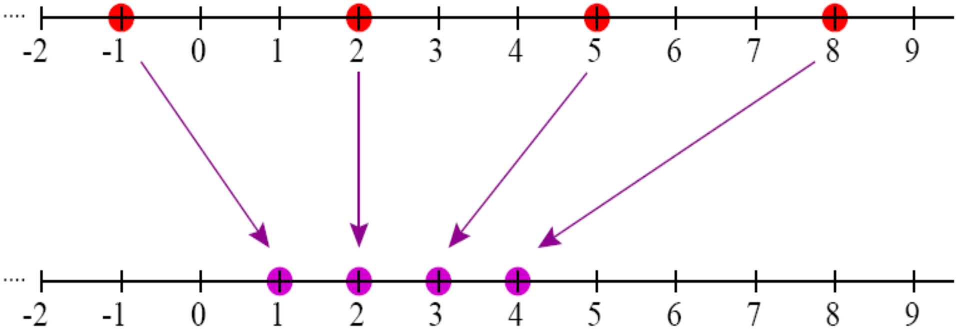
Data dependence graph:



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

→ 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
$I \leftarrow i * S - S + L$
```

Dependence Distance and Direction

Lexicographic order on iteration vectors → dynamic execution order:

$S_1(\langle i_1,\ldots,i_k\rangle) \lhd S_2(\langle j_1,\ldots,j_k\rangle)$ iff

either S_1 pred S_2 and $\langle i_1,\ldots,i_k\rangle) \leq_{lex} \langle j_1,\ldots,j_k\rangle$

or $S_1 = S_2$ and $\langle i_1,\ldots,i_k\rangle) <_{lex} \langle j_1,\ldots,j_k\rangle$

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

direction vector $dirv = \mathrm{sgn}(\vec{j} - \vec{i}) = \langle \mathrm{sgn}(j_1 - i_1), \ldots, \mathrm{sgn}(j_k - i_k)\rangle$

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)

Dependence Equation System

One-dimensional array A accessed in k nested loops:

$$S_1: \quad ...A[f(\vec{i})]...$$
$$S_2: \quad ...A[g(\vec{i})]...$$

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

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.,

$$\begin{array}{ll} 1 \le i_1 \le n_1, & 1 \le j_1 \le n_1, \\ \vdots & \vdots \\ 1 \le i_k \le n_k, & 1 \le j_k \le n_k \end{array}$$

iter. space constraints: linear inequalities

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

Linear Diophantine Equations

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

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

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 5

Theorem:

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

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 $i$ ...
    $S_1:\ \ldots A[x[i], 2*i]\ldots$
    $S_2:\ \ldots A[y[i], 2*i+1]\ldots$
```

```
for $i$ ...
    $S_1:\ \ldots A[i,i]\ldots$          $A[i*(s_1+1)]$
    $S_2:\ \ldots A[i,i+1]\ldots$        $A[i*(s_1+1)+1]$
```

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]

DF00100 Advanced Compiler Construction

TDDC86 Compiler optimizations and code generation

Loop Transformations and Parallelization

Christoph Kessler, IDA,
Linköping University

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 (→ DF00100)

Some important loop transformations

- Loop normalization
- Loop parallelization
- Loop invariant code hoisting
- Loop interchange
- Loop fusion vs. Loop distribution / fission
- Strip-mining / loop tiling / blocking vs. Loop linearization
- Loop unrolling, unroll-and-jam
- Loop peeling
- Index set splitting, Loop unswitching
- Scalar replacement, Scalar expansion
- Later: Software pipelining
- More: Cycle shrinking, Loop skewing, ...

Loop Invariant Code Hoisting

- **Move loop invariant code out of the loop**
  - Compilers can do this automatically *if* they can statically find out what code is loop invariant
  - Example:

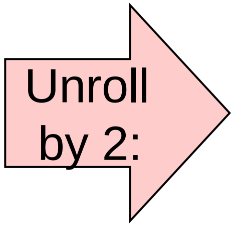
```
for (i=0;  i<10;  i++)
    a[i] = b[i]  + c / d;
```

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

Loop Unrolling

- **Loop unrolling**
  - Can be enforced with compiler options e.g. –funroll=2
  - Example:

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


Unroll by 2:

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

☺ Reduces loop overhead (total # comparisons, branches, increments)

☺ Longer loop body may enable further local optimizations
(e.g. common subexpression elimination,
register allocation, instruction scheduling,
using SIMD instructions)

☹ longer code

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

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
        $a[i] \leftarrow a[i] + b[j]$
    od
od
```

unroll&jam:

```
for $i$ from 1 to $N$ step 2 do
    for $j$ from 1 to $N$ do
        $a[i] \leftarrow a[i] + b[j]$
        $a[i+1] \leftarrow a[i+1] + b[j]$
    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
    $a[i] \leftarrow (x+y) * b[i]$
od
```

peel first iteration:

```
if $N \geq 1$ then
    $a[1] \leftarrow (x+y) * b[1]$
    for  $i$ from 2 to $N$ do
        $a[i] \leftarrow (x+y) * b[i]$
    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

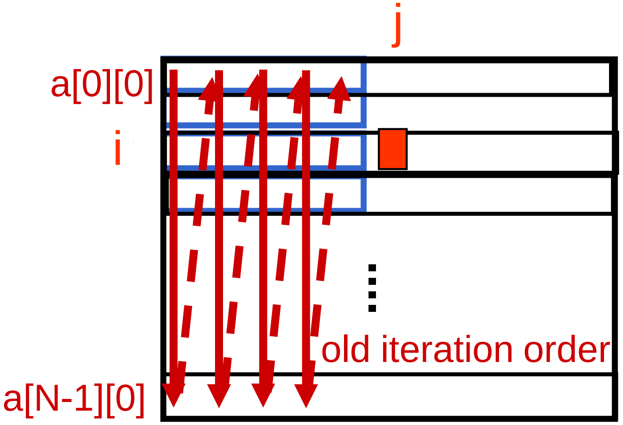
– longer code

Loop Interchange (1)

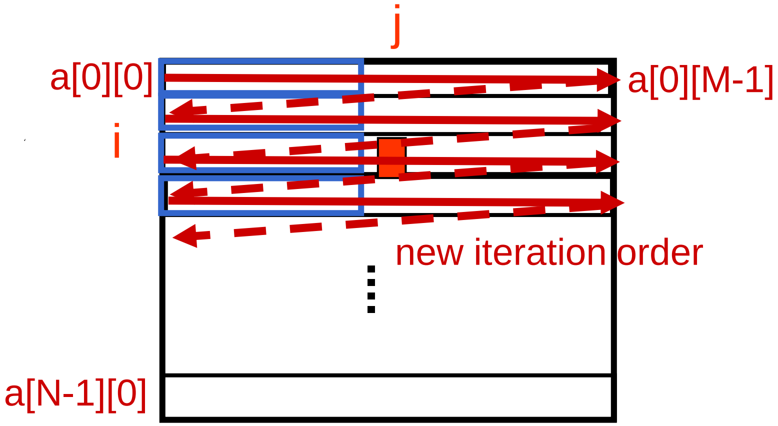
- For properly nested loops
  (statements in innermost loop body only)
  - Example 1:

```
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 ;
```



row-wise storage of 2D-arrays in C, Java



- Can improve data access locality in memory hierarchy (fewer cache misses / page faults)
- Can help with subsequent vectorization of innermost loops

Recall: Loop-Carried Data Dependences

- 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

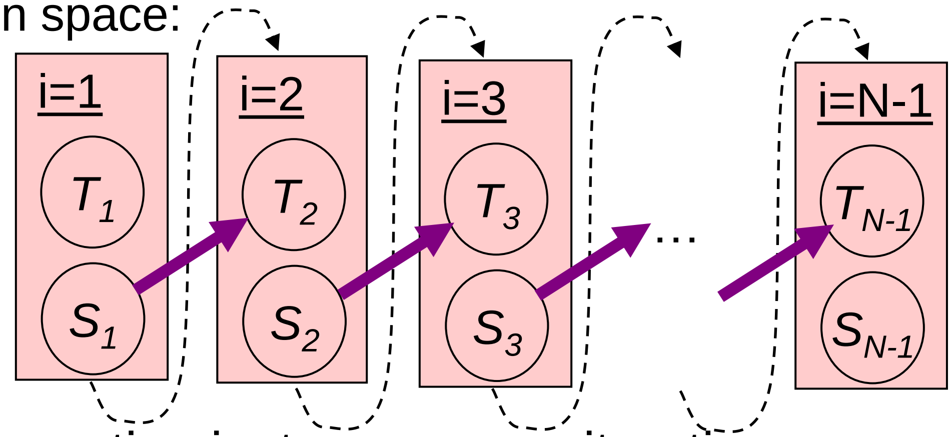
S: z = … ;
…
T: … = ..z.. ;

  - 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:

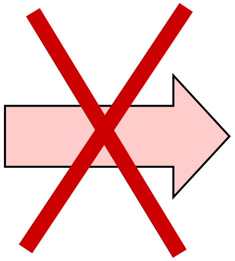


→ partial order between the operation instances resp. iterations

Loop Interchange (2)

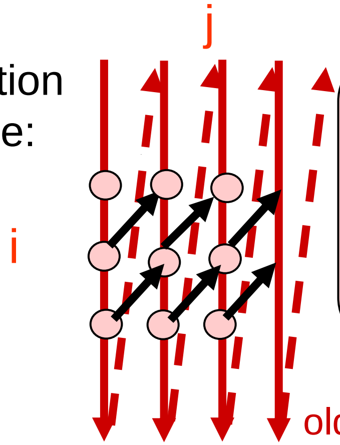
- **Be careful** with loop carried data dependences!
  - Example 2:

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



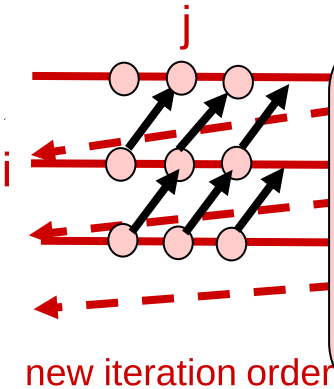
```
for (i=0;  i<N;  i++)
   for (j=1;  j<M;  j++)
      a[i][j] =…a[i+1][j-1]…;
```

Iteration space:



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

old iteration order



new iteration order

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

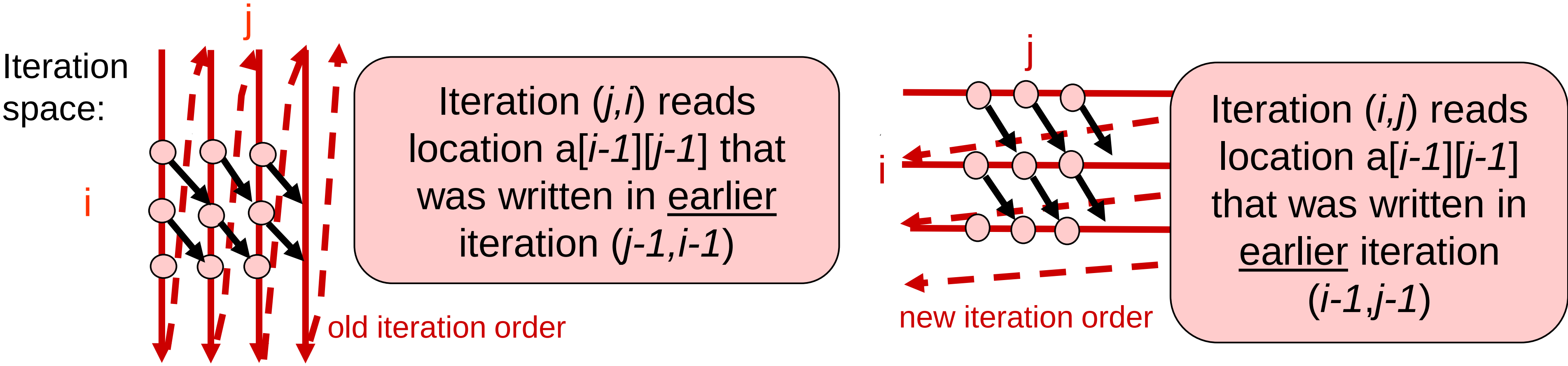
Loop Interchange (3)

- **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 (*j*,*i*) reads location a[*i-1*][*j-1*] that was written in <u>earlier</u> iteration (*j-1*,*i-1*)

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

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

Loop Fusion

- Merge subsequent loops with same header
  - Safe if neither loop carries a (backward) dependence
  - Example:

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

For N sufficiently large, a[*i*] will no longer be in the cache at this time

→

```
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 reduces number of branches

Loop Fusion
– Index variable name does not matter

```
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
```

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

fuse:

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

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

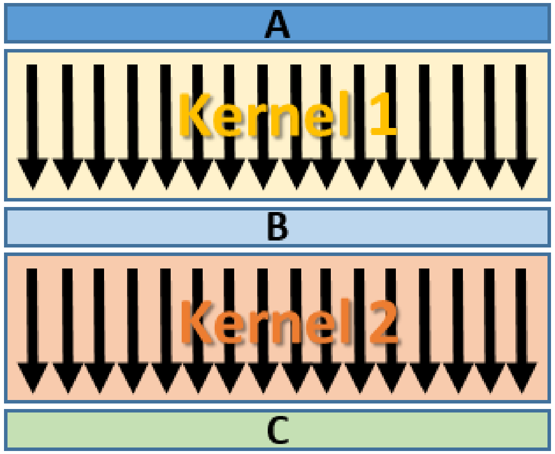
$j \leftarrow$ N (if downwards exposed)

Safe if neither loop carries a (backward) dependence.

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

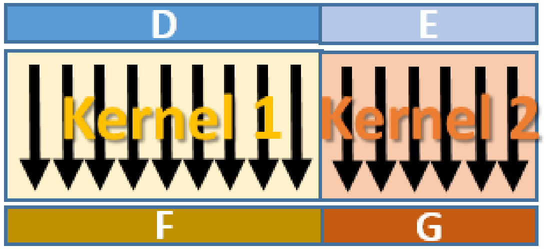
Special Case: Kernel Fusion for GPU

Serial Kernel Fusion



```
// start N1=N2 threads
{
    code_kernel1
    code_kernel2
}
```

Parallel Kernel Fusion



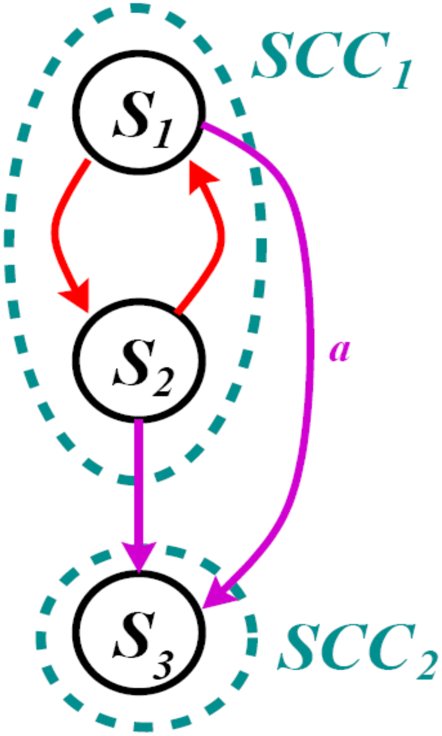
```
// start N1+N2 threads
{
   if (thread_idx < N1)
        code_kernel1
   else
        code_kernel2
}
```

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 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.

Example:

```
   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] ...              (=,<,=)
```

j-loop carries a dependence, its iteration order must be preserved

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.

Example:

```
   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] ...          (=,<,=)
```

j-loop carries a dependence, its iteration order must be preserved

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

Example:

```
   for (i=1; i<n; i++)
S:   b[i] = 2 * c[i];
```

Loop parallelization

```
forall ( i, 1, n, p )
   b[i] = 2 * c[i];
```

Principle: Parallelize outermost loop(s), vectorize innermost loop(s)

Remark on Loop Parallelization

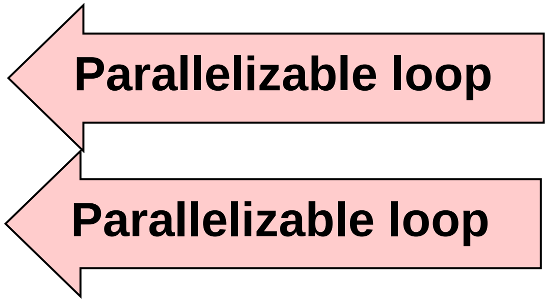
- Introducing temporary copies of arrays can remove some antidependences to enable automatic loop parallelization

- Example:

```
for (i=0; i<n; i++)
    a[i] = a[i] + a[i+1];
```

- The loop-carried dependence can be eliminated:

```
for (i=0; i<n; i++)
    aold[i+1] = a[i+1];
for (i=0; i<n; i++)
    a[i] = a[i] + aold[i+1];
```


Parallelizable loop

Parallelizable loop

Strip Mining / Loop Blocking

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

Loop blocking with block size *s*

```
for (ii=0; ii<n; ii+=s)     // loop over blocks
   for (i=ii; i<min(ii+s,n); i++)  // loop within block
      a[i] = b[i] + c[i];
```

Reverse transformation: Loop linearization

Loop (Nest) Tiling

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

Loop nest tiling with tile size *s* x *s* - **Step 1**: loop blocking

```
for (ii=0; ii<n; ii+=s)     // loop over blocks
  for (i=ii; i<min(ii+s,n); i++)  // loop within block
    for (jj=0; jj<m; jj+=s) // loop over blocks
      for (j=jj; j<min(jj+s,m); j++) // loop within blk
         a[i][j] = b[i][j] + c[j][i];
```

Loop (Nest) Tiling

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

Loop nest tiling with tile size *s* x *s* **- Step 2:** Loop interchange

```
for (ii=0; ii<n; ii+=s)      // loop over blocks
  for (jj=0; jj<m; jj+=s)    // loop over blocks
    for (i=ii; i<min(ii+s,n); i++) // loop within block
      for (j=jj; j<min(jj+s,m); j++) // loop within blk
        a[i][j] = b[i][j] + c[j][i];
```

Tiling = **loop blocking** for *multiple* loop headers in a loop nest
+ loop interchange
→ loops scanning a tile become innermost loops

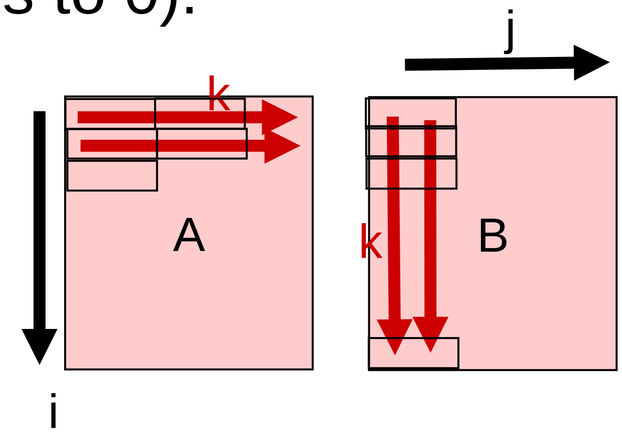
Goal: increase locality; support vectorization (vector registers)

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 ($\sim 10^3$):
  - $C_{ij} = \sum_{k=1..n} A_{ik} B_{kj}$ 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

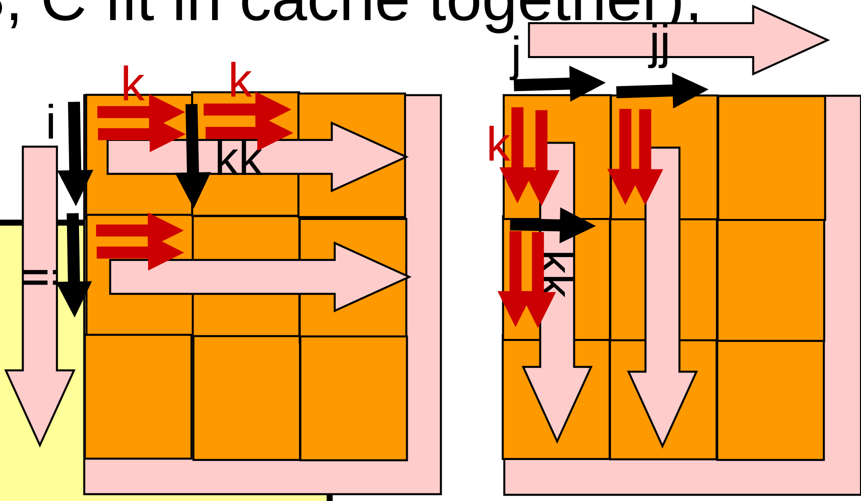
Bad spatial locality on B
(many capacity misses)

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];
```



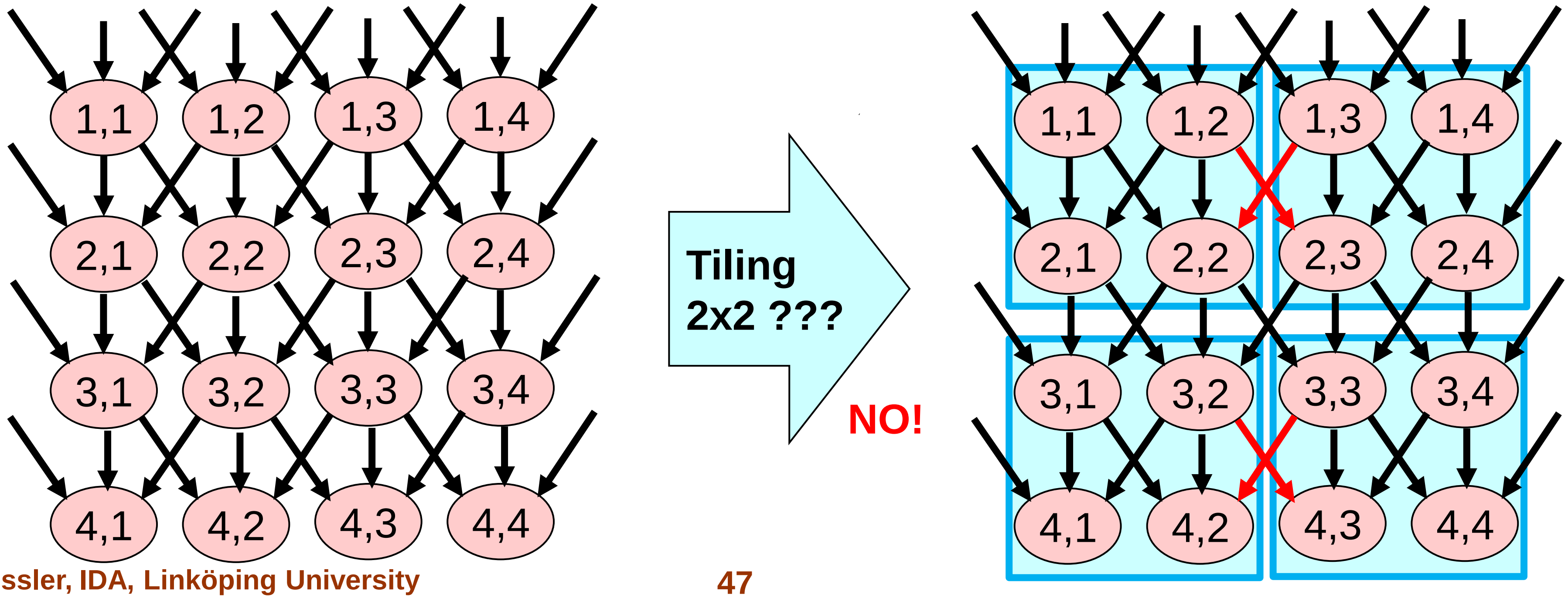
Good spatial locality for A, B and C

Loop (Nest) Tiling (cont.)

- Beware: Tiling is not always semantics-preserving
  - Dependences could lead to unschedulable code

Example:

```
        for i = 1, ..., 4
            for j = 1, ..., 4
S(i,j):          A[i][j] = x*A[i-1][j-1] + y*A[i-1][j] + z*A[i-1][j+1];
```



Tiling 2x2 ???

NO!

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
- In the best case, can make computations *cache-oblivious*
  - Performance largely independent of cache size

- **Further example**: AOS vs. SOA layout for images on CPU/GPU

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

Index Set Splitting

Divide the *iteration space* into two portions.

```
for $i$ from 1 to 100 do
    $a[i] \leftarrow b[i] + c[i]$
    if $i > 10$ then
        $d[i] \leftarrow a[i] + a[i-10]$
    fi
od
```

split after 10:

```
for $i$ from 1 to 10 do
    $a[i] \leftarrow b[i] + c[i]$
od
for $i$ from 11 to 100 do
    $a[i] \leftarrow b[i] + c[i]$
    $d[i] \leftarrow a[i] + a[i-10]$
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
    $a[i] \leftarrow a[i] + b[i]$
    if expression then
        $d[i] \leftarrow 0$
    fi
od
```

unswitch:

```
if expression then
    for $i$ from 1 to 100 do
        $a[i] \leftarrow a[i] + b[i]$
        $d[i] \leftarrow 0$
    od
else
    for $i$ from 1 to 100 do
        $a[i] \leftarrow a[i] + b[i]$
    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 Expansion / Array Privatization

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

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

expand scalar t:

```
if $N \geq 1$
    allocate $t'[1..N]$
    for $i$ from 1 to $N$ do
        $t'[i] \leftarrow a[i] + b[j]$
        $c[i] \leftarrow t'[i] + 1$
    od
    $t \leftarrow t'[N]$ // 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

Idiom recognition and algorithm replacement

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];
```

C. Kessler: Pattern-driven automatic parallelization. *Scientific Programming*, 1996

A. Shafiee-Sarvestani,
E. Hansson, C. Kessler: Extensible recognition of algorithmic patterns in DSP programs for automatic parallelization. *Int. J. on Parallel Programming*, 2013.

↓ Idiom recognition (pattern matching)

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

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

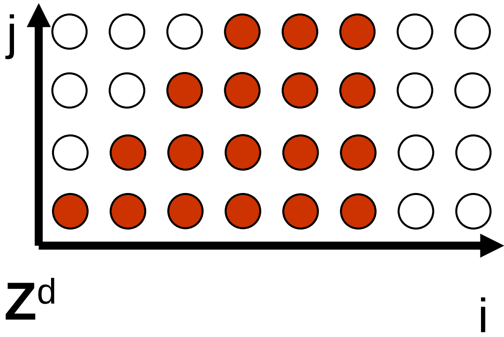
↓ Algorithm replacement

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

Polyhedral / Polytope Model

```
for i = 1 to N
   for j = 1 to min(i,M)
      loopbody( i, j )
```

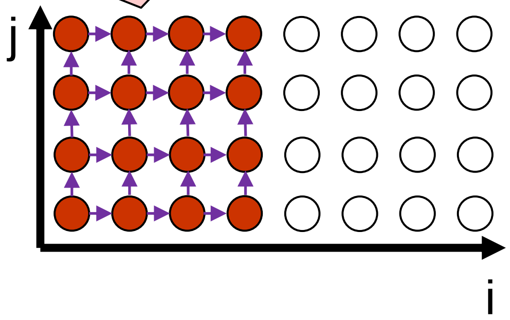
- Researched since late 1980s (with earlier roots), still active (see e.g. IMPACT workshop series)
- Compact representation of the **loop nest iteration space** of *d* perfectly nested loops as the points of a *polytope* (*polyhedron*) in $\mathbf{Z}^d$
  - Usually, loop normalization to obtain stride +1
  - E.g. in 2D: rectangular, triangular, trapezoidal, etc.



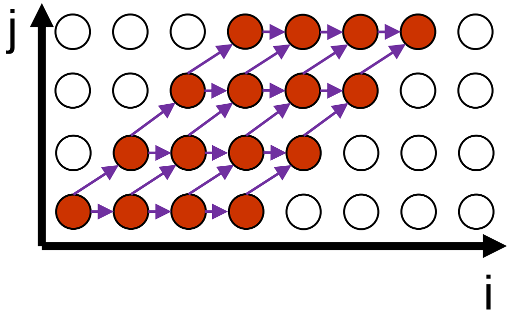
- Loop bounds must be **affine** (linear) functions of the indexes of outer loops (or constant)
  - The polytope is the intersection of halfspaces over $\mathbf{Z}^d$
  - The faces of the polytope are defined by the bounds of the loops
- Can apply described loop transformations as dependences allow
  - Can often be described as unimodular linear mappings
- **Parallelism** and scheduling options can be determined statically
  - constrained by the data dependences
- **Schedule** = space-time mapping of iterations to parallel processors and time axis must be affine.
- **Code generator** (e.g. cloog) generates code (nest of d for loops) that scans the polyhedron, given index bound parameters and a schedule

Polyhedral Example: Loop Nest Skewing and Parallelization

```
for i = 1 to N
    for j = 1 to M
        a[i,j] = f( a[i-1,j], a[i, j-1] )
```

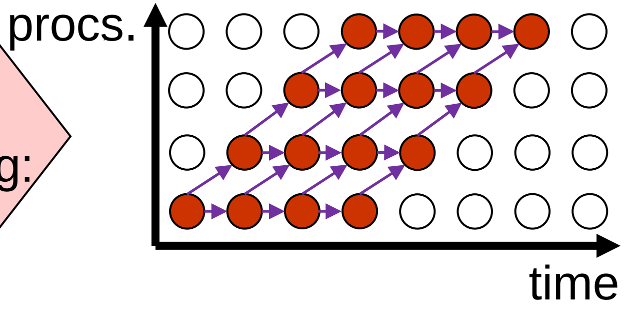


skewing:



(assuming here for simplicity that we have procs = N parallel processing units to use. If not, apply strip mining / tiling ...)

mapping/ scheduling:



generate HIR/src code:

```
forall proc = 1 to N
    for time = min(proc, N) to max(M+proc, M+N-1)
        a[i,j] = f( a[time-1, proc-1], a[time-1, proc] )
```

DF00100 Advanced Compiler Construction

TDDC86 Compiler optimizations and code generation

Concluding Remarks

Limits of Static Analyzability

Outlook: Runtime Analysis and Parallelization

Christoph Kessler, IDA,
Linköping University

Remark on static analyzability (1)

- Static dependence information is always a (safe) overapproximation of the real (run-time) dependences
  - Finding out the real ones exactly is statically undecidable!
  - If in doubt, a dependence must be assumed
    → may prevent some optimizations or parallelization
- One main reason for imprecision is **aliasing**, i.e. the program may have several ways to refer to the same memory location
  - Example: Pointer aliasing

```
void  mergesort ( int *a, int n )
{  …
   mergesort ( a,  n/2 );
   mergesort ( a + n/2, n-n/2 );
   …
}
```

How could a static analysis tool (e.g., compiler) know that the two recursive calls read and write disjoint subarrays of a?

Remark on static analyzability (2)

- Static dependence information is always a (safe) overapproximation of the real (run-time) dependences
  - Finding out the latter exactly is statically undecidable!
  - If in doubt, a dependence must be assumed
    → may prevent some optimizations or parallelization
- Another reason for imprecision are **statically unknown values** that imply whether a dependence exists or not
  - Example: Unknown dependence distance

```
// value of K statically unknown
for ( i=0; i<N; i++ )
{  …
   S:   a[i] = a[i] + a[K];
   …
}
```

Loop-carried dependence if K < N.
Otherwise, the loop is parallelizable.

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.

TDDC78 Programming of Parallel Computers

TDDD56 Multicore and GPU Programming

Run-Time Parallelization

Goal of run-time parallelization

- Typical target: **irregular loops**

  for (i=0; i<n; i++)
  a[i] = *f* (a[*g*(i)], a[*h*(i)], ...);

  - Array index expressions *g, h...* depend on run-time data
  - Iterations cannot be statically proved independent (and not either dependent with distance +1)

- **Principle:**
  At runtime, inspect *g, h* ... to find out the real dependences and compute a schedule for partially parallel execution
  - Can also be combined with speculative parallelization

Overview

- **Run-time parallelization of irregular loops**
  - DOACROSS parallelization
  - Inspector-Executor Technique (shared memory)
  - Inspector-Executor Technique (message passing) *
  - Privatizing DOALL Test *
- **Speculative run-time parallelization of irregular loops ***
  - LRPD Test *
- **General Thread-Level Speculation**
  - Hardware support *

* = not covered in this lecture. See the references.

DOACROSS Parallelization

- Useful if loop-carried dependence distances are unknown, but often > 1
- Allow independent subsequent loop iterations to overlap
- Bilateral synchronization between really-dependent iterations

Example:

```
for ( i=0; i<n; i++)
        a[i]  =  f ( a[ g(i) ], ... );
```

```
sh float aold[n];
sh flag done[n];   // flag (semaphore) array
forall i in 0..n-1   {   // spawn n threads, one per iteration
    done[i] = 0;
    aold[i] = a[i];    // create a copy
}
forall i in 0..n-1  {   // spawn n threads, one per iteration
    if (g(i) < i)   wait until done[ g(i) ] );
                    a[i]  =  f ( a[ g(i) ], ... );
                    set( done[i] );
    else
                    a[i]  =  f ( aold[ g(i) ], ... );  set done[i];
}
```

Inspector-Executor Technique (1)

- Compiler generates 2 pieces of customized code for such loops:

- **Inspector**
  - calculates values of index expression
    by simulating whole loop execution
    - typically, based on sequential version of the source loop (some computations could be left out)
  - computes implicitly the real iteration dependence graph
  - computes a **parallel schedule** as (greedy) wavefront traversal of the iteration dependence graph in topological order
    - all iterations in same wavefront are independent
    - schedule **depth** = #wavefronts = critical path length

- **Executor**
  - follows this schedule to execute the loop

Inspector-Executor Technique (2)

- **Source loop:**

```
 for ( i=0; i<n; i++)
         a[i]  =  f ( a[ g(i) ], a[ h(i) ], ... );
```

- **Inspector:**

```
int wf[n];  // wavefront indices
int depth = 0;
for (i=0; i<n; i++)
    wf[i] = 0;   // init.
for (i=0; i<n; i++) {
    wf[i] = max ( wf[ g(i) ], wf[ h(i) ], ... ) + 1;
    depth = max ( depth, wf[i] );
}
```

- Inspector considers only flow dependences (RAW), anti- and output dependences to be preserved by executor

Inspector-Executor Technique (3)

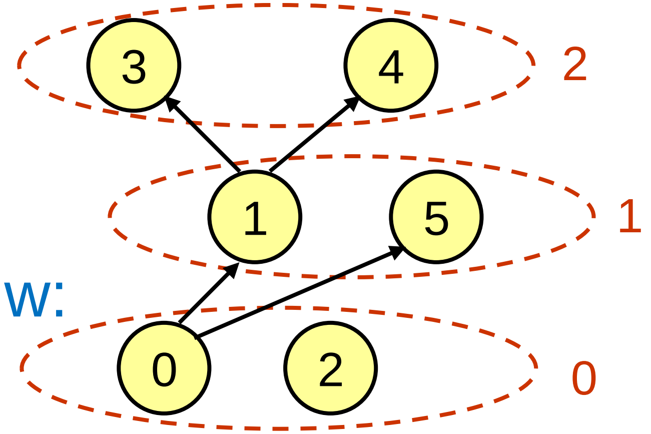
- **Example**:

```
for (i=0; i<n; i++)
      a[i] = ... a[ g(i) ] ...;
```

i	0	1	2	3	4	5
g(i)	2	0	2	1	1	0
wf[i]	0	1	0	2	2	1
g(i)<i ?	no	yes	no	yes	yes	yes

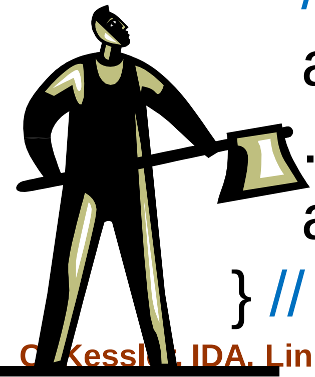
- **Executor**:

```
float aold[n];  // buffer array
aold[1:n] = a[1:n];
for (w=0; w<depth; w++)
   forall (i in {0..n-1}: wf[i] == w)  {
         // start task/thread where wf[i] == w:
         a1 = (g(i) < i)? a[g(i)] : aold[g(i)];
         ...  // similarly, a2 for h etc.
         a[i] =  f ( a1, a2, ... );
   } // wait for all threads of round w
```



iteration (flow) dependence graph

(depth=3)



Inspector-Executor Technique (4)

Problem: Inspector remains sequential – no speedup

Solution approaches:

- Re-use schedule over subsequent iterations of an outer loop if access pattern does not change
  - amortizes inspector overhead across repeated executions
- Parallelize the inspector using doacross parallelization [Saltz,Mirchandaney'91]
- Parallelize the inspector using sectioning [Leung/Zahorjan'91]
  - compute processor-local wavefronts in parallel, concatenate
  - trade-off schedule quality (depth) vs. inspector speed
  - Parallelize the inspector using bootstrapping [Leung/Z.'91]
  - Start with suboptimal schedule by sectioning, use this to execute the inspector → refined schedule

DF00100 Advanced Compiler Construction

TDDC86 Compiler optimizations and code generation

Thread-Level Speculation

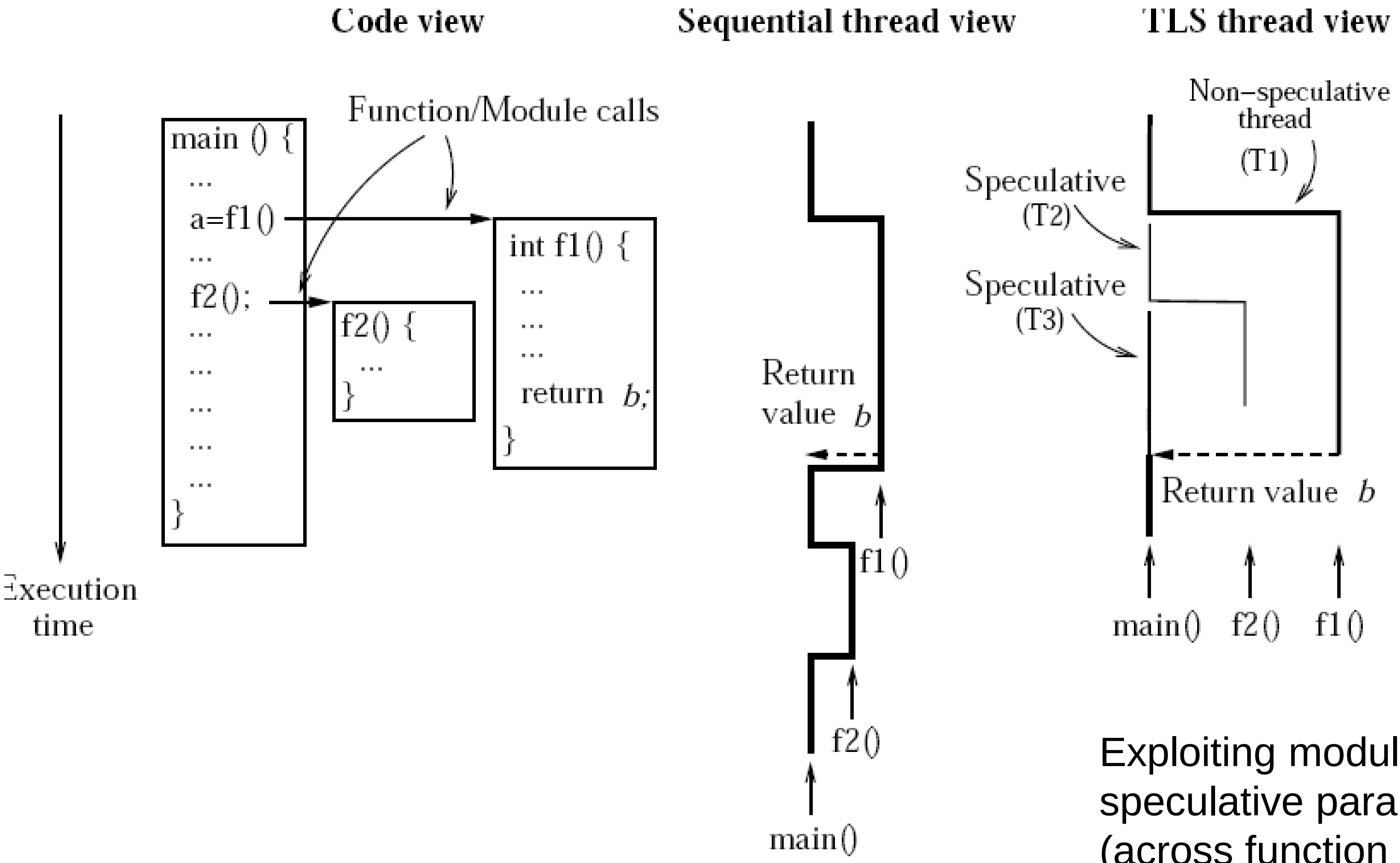
Christoph Kessler, IDA,
Linköping University

Speculatively parallel execution

- For automatic parallelization of sequential code where dependences are hard to analyze statically
- Works on a **task graph**
  - constructed implicitly and dynamically
- **Speculate on:**
  - control flow, data independence, synchronization, values

  We focus on thread-level speculation (TLS) for CMP/MT processors. Speculative instruction-level parallelism is not considered here.
- **Task:**
  - **statically:** Connected, single-entry subgraph of the control-flow graph
    - Basic blocks, loop bodies, loops, or entire functions
  - **dynamically:** Contiguous fragment of dynamic instruction stream within static task region, entered at static task entry

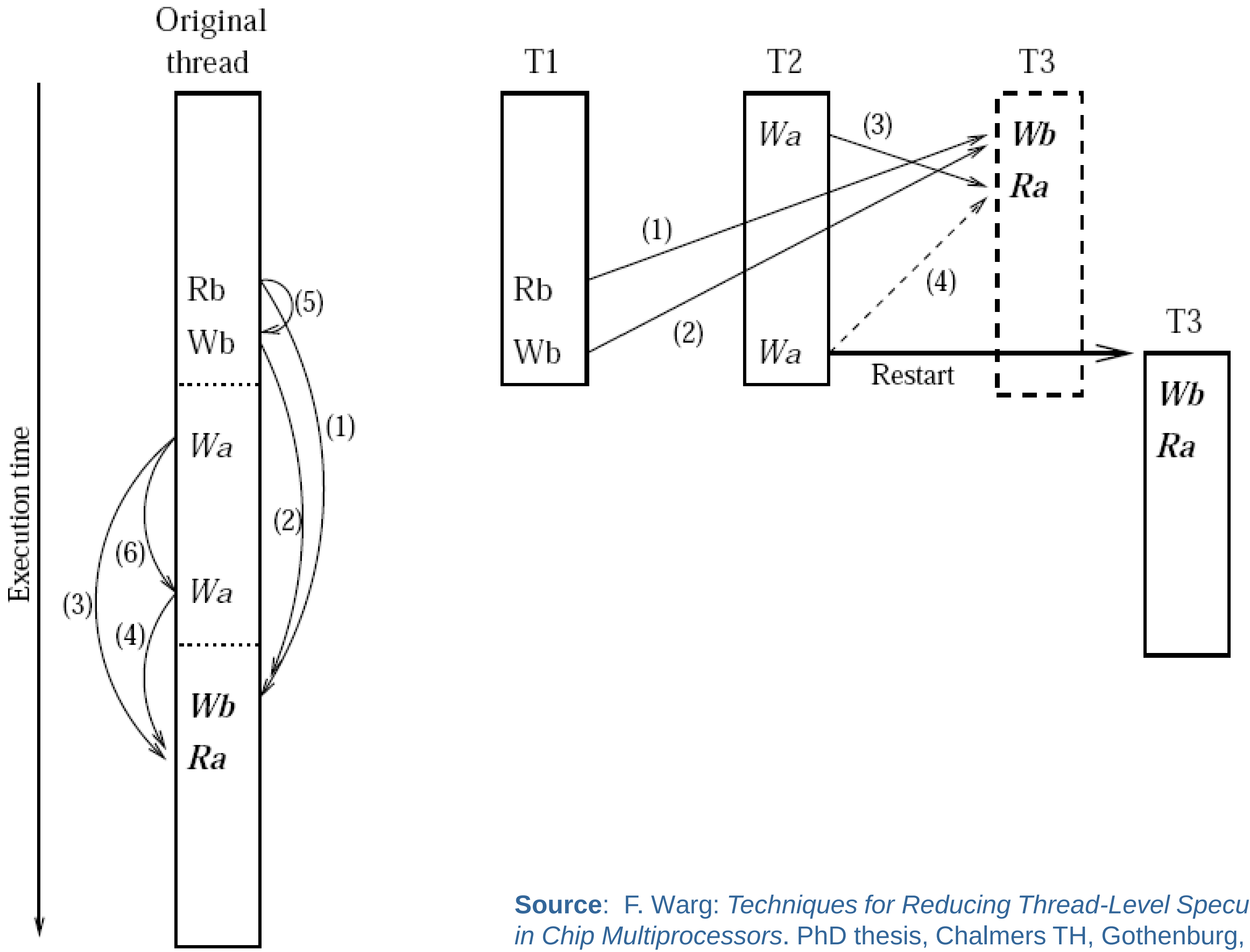
TLS Example



Exploiting module-level speculative parallelism (across function calls)

Source: F. Warg: *Techniques for Reducing Thread-Level Speculation Overhead in Chip Multiprocessors*. PhD thesis, Chalmers TH, Gothenburg, June 2006.

Data dependence problem in TLS



Source: F. Warg: *Techniques for Reducing Thread-Level Speculation Overhead in Chip Multiprocessors*. PhD thesis, Chalmers TH, Gothenburg, June 2006.

Speculatively parallel execution of tasks

- **Speculation on inter-task control flow**
  - After having assigned a task,
    predict its successor task and start it speculatively
- **Speculation on data independence**
  - For inter-task memory data (flow) dependences
    - conservatively: await write (memory synchronization, message)
    - speculatively: hope for independence and continue (execute the load)
- **Roll-back** of speculative results on mis-speculation (expensive)
  - When starting speculation, state must be buffered
  - Squash an offending task and all its successors, restart
- **Commit speculative results** when speculation resolved to correct
  - Task is retired

Selecting Tasks for Speculation

- **Small tasks:**
  - too much overhead (task startup, task retirement)
  - low parallelism degree
- **Large tasks:**
  - higher misspeculation probability
  - higher rollback cost
  - many speculations ongoing in parallel may saturate the resources
- **Load balancing issues**
  - avoid large variation in task sizes
- Traversal of the program’s control flow graph (CFG)
  - Heuristics for task size, control and data dep. speculation

TLS Implementations

- **Software-only speculation**
  - for loops [Rauchwerger, Padua ’94, ’95]
  - ...

- **Hardware-based speculation**
  - Typically, integrated in cache coherence protocols
  - Used with multithreaded processors / chip multiprocessors for automatic parallelization of sequential legacy code
  - If source code available, compiler may help e.g. with identifying suitable threads

Some references on Dependence Analysis, Loop optimizations and Transformations

- H. Zima, B. Chapman: *Supercompilers for Parallel and Vector Computers*. Addison-Wesley / ACM press, 1990.
- M. Wolfe: *High-Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.
- R. Allen, K. Kennedy: *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2002.

Idiom recognition and algorithm replacement:

- C. Kessler: Pattern-driven automatic parallelization. *Scientific Programming* **5**:251-274, 1996.
- A. Shafiee-Sarvestani, E. Hansson, C. Kessler: Extensible recognition of algorithmic patterns in DSP programs for automatic paral-lelization. *Int. J. on Parallel Programming*, 2013.

Questions?

Some references on Dependence Analysis, Loop optimizations and Transformations

- H. Zima, B. Chapman: *Supercompilers for Parallel and Vector Computers*. Addison-Wesley / ACM press, 1990.
- M. Wolfe: *High-Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.
- R. Allen, K. Kennedy: *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2002.

Idiom recognition and algorithm replacement:

- C. Kessler: Pattern-driven automatic parallelization.
  Scientific Programming **5**:251-274, 1996.
- A. Shafiee-Sarvestani, E. Hansson, C. Kessler:
  Extensible recognition of algorithmic patterns in DSP programs for automatic parallelization. *Int. J. on Parallel Programming*, 2013.

Frameworks

- Polly
- Cloog
- PluTo polyhedral transformation framework:
  An automatic parallelizer and locality optimizer for affine loop nests
  http://pluto-compiler.sourceforge.net/

Polyhedral Compilation Frameworks

- Closely related to (parametric) integer programming
  - PIPS, PIPlib
  - Paul Feautrier: Dataflow Analysis of Array and Scalar References. International Journal of Parallel Programming, 1991
- and many others

More recent work e.g.

- Polly for LLVM: https://polly.llvm.org/
- PluTo
  - U. Bondhugula, PhD thesis, 2008: https://www.csa.iisc.ac.in/~udayb/publications/uday-thesis.pdf
- Cloog
  - for code generation (scanning a polyhedron, given iteration domain bounds and a schedule)
  - http://www.cloog.org
- Polybench polyhedral benchmark suite
- Annual IMPACT workshop series at HiPEAC conference

Some references on run-time parallelization

- R. Cytron: Doacross: Beyond vectorization for multiprocessors. Proc. ICPP-1986

- D. Chen, J. Torrellas, P. Yew: An Efficient Algorithm for the Run-time Parallelization of DO-ACROSS Loops, Proc. IEEE Supercomputing Conf., Nov. 2004, IEEE CS Press, pp. 518-527

- R. Mirchandaney, J. Saltz, R. M. Smith, D. M. Nicol, K. Crowley: Principles of run-time support for parallel processors, Proc. ACM Int. Conf. on Supercomputing, July 1988, pp. 140-152.

- J. Saltz and K. Crowley and R. Mirchandaney and H. Berryman: Runtime Scheduling and Execution of Loops on Message Passing Machines, *Journal on Parallel and Distr. Computing* 8 (1990): 303-312.

- J. Saltz, R. Mirchandaney: The preprocessed doacross loop. Proc. ICPP-1991 Int. Conf. on Parallel Processing.

- S. Leung, J. Zahorjan: Improving the performance of run-time parallelization. Proc. ACM PPoPP-1993, pp. 83-91.

- Lawrence Rauchwerger, David Padua: The Privatizing DOALL Test: A Run-Time Technique for DOALL Loop Identification and Array Privatization. Proc. ACM Int. Conf. on Supercomputing, July 1994, pp. 33-45.

- Lawrence Rauchwerger, David Padua: The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. Proc. ACM SIGPLAN PLDI-95, 1995, pp. 218-232.

Some references on speculative execution / parallelization

- T. Vijaykumar, G. Sohi: Task Selection for a Multiscalar Processor. Proc. MICRO-31, Dec. 1998.
- J. Martinez, J. Torrellas: Speculative Locks for Concurrent Execution of Critical Sections in Shared-Memory Multiprocessors. Proc. WMPI at ISCA, 2001.
- F. Warg and P. Stenström: Limits on speculative module-level parallelism in imperative and object-oriented programs on CMP platforms. Pr. IEEE PACT 2001.
- P. Marcuello and A. Gonzalez: Thread-spawning schemes for speculative multithreading. Proc. HPCA-8, 2002.
- J. Steffan et al.: Improving value communication for thread-level speculation. HPCA-8, 2002.
- M. Cintra, J. Torrellas: Eliminating squashes through learning cross-thread violations in speculative parallelization for multiprocessors. HPCA-8, 2002.
- Fredrik Warg and Per Stenström: Improving speculative thread-level parallelism through module run-length prediction. Proc. IPDPS 2003.
- F. Warg: *Techniques for Reducing Thread-Level Speculation Overhead in Chip Multiprocessors*. PhD thesis, Chalmers TH, Gothenburg, June 2006.
- T. Ohsawa et al.: Pinot: Speculative multi-threading processor architecture exploiting parallelism over a wide range of granularities. Proc. MICRO-38, 2005.