Optimization and Parallelization of Sequential Programs

Lecture 7

Christoph Kessler
IDA / PELAB
Linköping University
Sweden

Outline

Towards (semi-)automatic parallelization of sequential programs

- Data dependence analysis for loops
- Some loop transformations
  - Loop invariant code hoisting, loop unrolling, loop fusion, loop interchange, loop blocking and tiling
- Static loop parallelization
- Run-time loop parallelization
  - Doacross parallelization, Inspector-executor method
- Speculative parallelization (as time permits)
- Auto-tuning (later, if time)

Foundations: Control and Data Dependence

- Consider statements \( S, T \) in a sequential program (\( S \Rightarrow T \) possible)
  - Scope of analysis is typically a function, i.e. intra-procedural analysis
  - Assume that a control flow path \( S \ldots T \) 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 \( S \Rightarrow T \)
  - If the fact whether \( T \) is executed may depend on \( S \) (e.g. condition)
  - Implies that relative execution order \( S \Rightarrow T \) 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)

- Data dependence \( S \Rightarrow T \)
  - If statement \( S \) may execute (dynamically) before \( T \) and both may access the same memory location
  - Means that execution order \( S \Rightarrow T \) 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

Dependence Graph

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

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
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) criterion: preserve data dependences
- Need data dependence analysis for loops

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

\[ S_1 \text{ statically (textually) precedes } S_2 \quad \text{ or } \quad S_1 \text{ pred } S_2 \]
\[ S_1 \text{ dynamically precedes } S_2 \quad \text{ or } \quad S_1 < S_2 \]

Within loops, loop nests:
\[ \text{pred} \neq \varnothing \]

\[ S_1 : \quad \text{for } i \text{ from } 1 \text{ to } n \text{ do} \]
\[ S_2 : \quad t \leftarrow a[i] \]
\[ S_3 : \quad a[i+1] \leftarrow t \]
\[ \text{od} \]

Loop Iteration Space

Beyond basic blocks: \( \text{pred} \neq \varnothing \)

Canonical loop nest: (III code)

\[ \text{for } i_1 \text{ from } 1 \text{ to } n_1 \text{ do} \]
\[ \quad \text{for } i_2 \text{ from } 1 \text{ to } n_2 \text{ do} \]
\[ \quad \text{for } i_k \text{ from } 1 \text{ to } n_k \text{ do} \]
\[ S_{(i_1, \ldots, i_k)} : \quad A[i_1, 2 \cdot i_2 + i_3] \leftarrow B[i_1, i_2] + 1 \]
\[ S_{(i_1, \ldots, i_k)} : \quad B[i_1, i_2] + i_3 \leftarrow 2 \cdot A[i_1, 2 \cdot i_2 + i_3] \]

Iteration space: \( BS = [1 \cdot n_1] \times [1 \cdot n_2] \times \ldots \times [1 \cdot n_k] \)

(Example)

\[ (i_1, \ldots, i_k) \in BS \]

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

For \( i \) from 2 to 9 do
\[
S_1: X[i] \gets Y[i] + Z[i]
\]
\[
S_2: A[i] \gets A[i-1] + 1
\]
\[
\text{od}
\]

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

There is a loop-carried, forward, flow dependence from \( S_1 \) to \( S_2 \).

Data dependence graph:

Loop Normalization

Given a loop of the form

\[
\text{for } i \text{ from } L \text{ to } U \text{ step } D \text{ do}
\]
\[
\text{od}
\]

normalize the loop:

- lower bound \( (C) \) resp. \( 1 \) (Fortran)
- step size \( \pm 1 \)

\[
\text{update all occurrences of the loop counter } i \text{ by } i \gets i + D
\]

for \( i \) from 1 to \((U-L+D)/D \) step 1 do
\[
\text{od}
\]

\[
I \gets i + D \quad S \gets L
\]

Dependence Distance and Direction

Lexicographic order on iteration vectors \( \rightarrow \) dynamic execution order:

\[
S_1(i) < S_2(i) \text{ iff either } S_1(i) \text{ pred } S_2(i) \text{ or } S_1(i) = S_2(i) \text{ and } (i_1, i_2, \ldots, i_N) <_{\text{lex}} (j_1, j_2, \ldots, j_N)
\]

Distance vector \( d = j - i = (j_1 - i_1, \ldots, j_N - i_N) \)

Direction vector \( d_{\text{dir}} = \text{sgn}(d) = (\text{sgn}(j_1 - i_1), \ldots, \text{sgn}(j_N - i_N)) \)

in terms of symbolic \( < > \leq \geq \)

Example:

\[
S_1((i_1, i_2, i_3)) < S_2((j_1, j_2, j_3))
\]

distance vector \( d = (0, 0, 0) \), direction vector \( d_{\text{dir}} = (-1, -1, -1) \),

loop-independent dependence

Example:

\[
S_1((i_1, i_2, i_3)) \not< S_2((j_1, j_2, j_3))
\]

distance vector \( d = (0, 0, 0) \), direction vector \( d_{\text{dir}} = (0, 0, 0) \),

loop-carried dependence (carried by \( j \) loop at level 3)

Linear Diophantine Equations

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

where \( n \geq 1 \), \( a_j, c \in \mathbb{Z}, \exists j: a_j \neq 0, x_j \in \mathbb{Z} \)

Example 1: \( x + 4y = 1 \)

has infinitely many solutions, e.g. \( x = 1 \) and \( y = -1 \).

Example 2: \( 5x - 10y = 2 \)

has no solution in \( \mathbb{Z} \); absolute term must be multiple of \( 10 \).

Theorem:

\[
\sum_{j=1}^{n} a_j x_j = c \text{ has a solution iff } \gcd(a_1, a_2, \ldots, a_n) \mid c.
\]

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

Dependence Testing, 1: GCD-Test

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

\[
gcd \left( \bigcup_{i=1}^{m} (a_i, b_i) \right) \mid \sum_{i=1}^{m} (a_i - b_i)
\]

constraints on \( b \text{S} \) not considered

Example:

For \( i \) from 1 to 4 do
\[
S_1: b[i] := a[3 * i - 5] + 2
\]
\[
S_2: a[2 * i + 1] := 10 \times i
\]

solution to \( 2i + 1 = 3j - 5 \) exists for \( j \in \mathbb{Z} \) as \( \gcd(3, 2) \mid (-5 - 1 + 3 - 2) \)

not checked whether such \( i, j \) exist in \( \{1, \ldots, 4\} \).
**For multidimensional arrays?**

Subscript-wise test vs. linearized indexing:

```
for i...
S1: ... A[i][j][2*i+j]...
S2: ... A[i][j][2*i+1+j]...
```

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 [Griff/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 Transformations and Parallelization**

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) criterion: preserve data dependences
- Need data dependence analysis for loops

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

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<N; i++) {
        a[i] = b[i];
    }
    ```
    → 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

---

**Foundations:**

**Loop-Carried Data Dependences**

- **Recall:** Data dependence $S \rightarrow T$.
  - 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 $S \rightarrow T$ is called loop carried by a loop $L$**
  - If the data dependence $S \rightarrow T$ may exist for instances of $S$ and $T$ in different iterations of $L$.
    - Example:
      ```
      L: for (i=0; i<N; i++) { S: z = ...; T: ... = z; }
      ```
      → partial order between the operation instances resps. iterations

---

**Loop Interchange (1)**

- **For properly nested loops**
  - (statements in innermost loop body only)
    - Example 1:
      ```
      for (i=0; i<N; i++) { a[i][j] = 0.0; }
      for (j=0; j<M; j++) { a[i][j] = 0.0; }
      ```
      → Can improve data access locality in memory hierarchy
        (fewer cache misses / page faults)

- **Example:**
  ```
  for (i=0; i<N; i++) {
    ... a[i][j] = ...;
  }
  for (j=0; j<M; j++) {
    ... a[i][j] = ...;
  }
  ```
  → Can improve data access locality in memory hierarchy
    (fewer cache misses / page faults)

---

**Loop Interchange (2)**

- **Be careful** with loop carried data dependences!
  - Example 2:
    ```
    for (i=0; i<N; i++) {
      a[i][j] = ...;
    }
    for (j=0; j<M; j++) {
      a[i][j] = ...;
    }
    ```
    → 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 (i=1; i<M; i++)
    for (j=1; j<M; j++)
    a[i][j] = ...;
    ```
    → 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
  - Safe if neither loop carries a (backward) dependence
  - Example:
    ```
    for (i=0; i<N; 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

---

**Exercise:** Formulate the unrolling rule for statically unknown upper loop limit
Loop Iteration Reordering

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

Example:

\[
\text{for } (i=0; i<n; i++) \\
\text{ for } (j=0; j<n; j++) \\
\text{ for } (k=0; k<n; k++) \\
C[i][j] = A[i][k] * B[k][j];
\]

Loop Parallelization

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

Example:

\[
\text{for } (i=0; i<n; i++) \\
\text{ for } (j=0; j<n; j++) \\
\text{ for } (k=0; k<n; k++) \\
C[i][j] = A[i][k] * B[k][j];
\]

Remark on Loop Parallelization

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

Example:

\[
\text{for } (i=0; i<n; i++) \\
a[i] = a[i] + a[i+1];
\]

The loop-carried dependence can be eliminated:

\[
\text{for } (i=0; i<n; i++) \\
aold[i+1] = a[i+1];
\]

Parallelizable loop

Parallelizable loop

Tiled Matrix-Matrix Multiplication (1)

- Matrix-Matrix multiplication \( C = A \times B \) here for square \( (n \times n) \) matrices \( A, B, C \), with \( n \) large (-10^3):
  \[
  C_{ij} = \sum_{k=0}^{n-1} A_{ik} B_{kj} \quad \text{for all } i, j = 1...n
  \]

- Standard algorithm for Matrix-Matrix multiplication (here without the initialization of \( C \)-entries to 0):

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:

\[
\text{for } (ii=0; ii<n; ii++) \\
\text{ for } (jj=0; jj<n; jj++) \\
\text{ for } (kk=0; kk<n; kk++) \\
C[i][j] += A[i][k] * B[k][j];
\]

Strip Mining / Loop Blocking / -Tiling

- Loop blocking with block size \( S \)

\[
\text{for } (i=0; i<n; i++) \\
\text{ for } (j=0; j<n; j++) \\
\text{ for } (k=0; k<n; k++) \\
C[i][j] += A[i][k] * B[k][j];
\]

Good spatial locality on A, C
Bad spatial locality on B (many capacity misses)
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

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

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

Finding the best layout for all multidimensional arrays is a NP-complete optimization problem.

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

Loop Fusion

```
for i from 1 to N do
    c[i] += a[i] + b[i]
od
for i from 1 to N do
    a[i] *= c[i]
    b[i] += a[i]
od
```

Safe if neither loop carries a (backward) dependence.
- locality: can convert inter-loop reuse to intra-loop reuse
- larger basic blocks
- reduce loop overhead

Loop Nest Flattening / Linearization

```
for i from 0 to \( n-1 \) do
    for j from 0 to \( m-1 \) do
        i = k \( \mod \) m
        j = k \( \mod \) m
        od
    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 step 4 do
    a[i] += a[i] + b[i]
    a[i+1] += a[i+1] + b[i+1]
    a[i+2] += a[i+2] + b[i+2]
    a[i+3] += a[i+3] + b[i+3]
od
```

- less overhead per useful operation
- larger basic blocks for local optimizations
  - (local CSE, local reg-utilization, local scheduling, SW pipelining)
- shorter code

Loop Unrolling with Unknown Upper Bound

```
for i from 1 to N do
    a[i] += a[i] + b[i]
    a[i+1] += a[i+1] + b[i+1]
    a[i+2] += a[i+2] + b[i+2]
    a[i+3] += a[i+3] + b[i+3]
    i = i + 4
od
```

```
while i < N do
    a[i] += a[i] + b[i]
    i = i + 1
od
```

used e.g. in RI AS
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
$a[i] \leftarrow a[i] + b[j]$
end for
end for

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

+ increases reuse in inner loop
+ less overhead

Index Set Splitting

Divide the iteration space into two portions.

for $i$ from 1 to 100 do
$a[i] \leftarrow b[i] + c[i]$
end for

if $i > 10$ then
$a[i] \leftarrow a[i] + a[i - 10]$
end if

split after 10:

$\downarrow$

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

Loop Peeling

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

if $N \geq 1$ then
$a[i] \leftarrow (x + y) * b[i]$
end if

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

if $N \geq 1$ then
$a[i] \leftarrow (x + y) * b[i]$
end if

Loop Unswitching

if expression then
for $i$ from 1 to 100 do
$a[i] \leftarrow a[i] + b[i]$
end for

else
for $i$ from 1 to 100 do
$a[i] \leftarrow a[i] + b[i]$
end for

split after 10:

$\downarrow$

+ hoist loop invariant control flow out of loop nest
+ no tests, no branches in loop body
+ larger basic blocks (so 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 $N$ step 2 do
for $j$ from 1 to $N$ do
$a[i] \leftarrow a[i] + b[j]$
end for
end for

$\downarrow$

+ keep $i$ 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
$c[i] \leftarrow c[i] + b[i]$
end for

$\downarrow$

+ removes the loop-carried antidependence due to $i$
+ 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 $i$ for every processor = array privatization
Idiom recognition and algorithm replacement

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

\[
\begin{align*}
S0: & \quad s = 0; \\
S1: & \quad s = s + a[1]; \\
S2: & \quad a[0] = c[0]; \\
S3: & \quad a[i] = a[i-1] \times b[i] + c[i];
\end{align*}
\]

\(\text{Idiom recognition (pattern matching)}\)

\[
\begin{align*}
S1': & \quad s = \text{VSUM}(a[1:n-1], 0); \\
S1'': & \quad a[0:n-1] = \text{FOLR}(b[1:n-1], c[0:n-1], \text{mul}, \text{add});
\end{align*}
\]

\(\text{Algorithm replacement}\)

**Concluding Remarks**

Limits of Static Analyzability

**Outlook: Runtime Analysis and Parallelization**

**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

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

**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

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

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

**Outlook: Runtime Parallelization**

Sometimes parallelizability cannot be decided statically.

```java
if \_\_\_parallelizable( )
    for all 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.
Goal of run-time parallelization

- Typical target: irregular loops
  ```c
  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

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:
```c
for (i=0; i<n; i++)
  a[i] = f(a[g(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
  - Executor
    - Follows this schedule to execute the loop

Inspector-Executor Technique (2)

- Source loop:
  ```c
  for (i=0; i<n; i++)
  a[i] = ... a[g(i)] ...;
  ```
- Inspector:
  ```c
  int wf[n]; // wavefront indices
  int depth = 0;
  for (i=0; i<n; i++)
    wf[i] = 0; // init.
    for (w=0; w<depth; w++)
      for all i, 0..n-1
        if (wf[i] == w)
          a[i] = f(aold[g(i)]);
          ... // similarly, a2 for h etc.
          a[i] = f(a1, a2, ...);
  ```

Inspector-Executor Technique (3)

- Example:
  ```c
  for (i=0; i<n; i++)
  a[i] = ... a[g(i)] ...;
  ```
- Executor:
  ```c
  float aold[n]; // buffer array
  aold[1:n] = a[1:n];
  for (w=0; w<depth; w++)
    for all (i, 0, n, #)
      if (wf[i] == w)
        a[i] = f(a1, a2, ...);
  ```

Hardware support

* = not covered in this course. See the references.
Inspector-Executor Technique (4)

Problem: Inspector remains sequential – no speedup

Solution approaches:
- Re-use schedule over subsequent iterations of an outer loop if access patterns do not change
- Parallelize the inspector using doacross parallelization
  [Saltz/Mirchandaney’91]
- Parallelize the inspector using sectioning
  [Leung/Zahorjan’91]
- Parallelize the inspector using bootstrapping
  [Leung/Z.’91]
- Start with suboptimal schedule by sectioning, use this to execute the inspector → refined schedule

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

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

Data dependence problem in TLS
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

Questions?

Some references on Dependence Analysis, Loop optimizations and Transformations


Idiom recognition and algorithm replacement:


Some references on run-time parallelization


Some references on speculative execution / parallelization