A Catalog of Transformations

Name	Purpose	Benefit
	Preprocessing transformations	
Loop normalization	Make loops canonical	Simplify,improve dep. analysis
Ind. var. substitution	Identify aux. induction vars	Improve dependence information
Scalar expansion	Replace scalar with array	Eliminate spurious dependences
Scalar/array privatization	Treat var. as iteration-private	Eliminate spurious dependences
Variable renaming	Use multiple copies of vars	Eliminate anti- and output-dependences
Reduction recognition	Recognize reductions	Ignore special-case dependences
	Reordering transformations	
Loop interchange	Change loop nesting order	Cache, parallelism, vectorization
Loop strip-mining	Make 2 nested loops	”
Loop skewing	Change wavefront loop to parallel	Improve loop parallelism
Loop reversal	Run loop backwards	Reduce array storage
Index set splitting	Break loop by index space	Remove some deps.
Loop distribution	Break loop by statements	Simplify parallelization, vectorization
Loop alignment	Change carried to indep.	Simplify parallelization, vectorization
Loop fusion	Join loops by statements	Improve cache reuse

Optimizations Using the Loop Transformations

Memory hierarchy optimizations

Transformations?
Goal 1: Improving reuse of data values within loop nest
Goal 2: Exploit reuse to reduce cache, TLB misses

Tiling

Transformations?
Goal 1: Exploit temporal reuse when data size > cache size
Goal 2: In parallel loops, reduce synch'n, comm'n overhead

Software Prefetching

Transformations?
Goal: Prefetch predictable accesses$k$ iterations ahead

Optimizations Using the Loop Transformations (cont'd)

Software Pipelining

Transformations?
Goal: Extract ILP from multiple consecutive iterations

Automatic parallelization

Also, auto-vectorization

Transformations?
Goal 1: Enhance parallelism
Goal 2: Convert scalar loop to explicitly parallel
Goal 3: Improve performance of parallel code

Loop Interchange

Informal Definition

Change nesting order of loops in a perfect loop nest, with no other changes.

Examples

```
do i=2, N
    do j=2, M-1
        A[i,j] = A[i,j] * 2
    enddo
enddo
```

```
do j=2, M-1
    do i=2, N
        A[i,j] = A[i,j] * 2
    enddo
enddo
```

```
do i=2, N
    do j=2, M-1
        A[i,j] = ... * B[i-1,j-1]
        B[i,j] = ... + A[i,j] + A[i-1,j]
    enddo
enddo
```

```
do j=2, M-1
    do i=2, N
        A[i,j] = ...
        B[i,j] = ...
    enddo
enddo
```

Loop Interchange

Uses

1. Move independent loop innermost
2. Move independent loop outermost
3. Make accesses stride-1 in memory
4. Loop tiling (combine with strip-mining)
5. Unroll-and-jam (combine with unrolling)
6. Many others

Direction Vectors and Loop Interchange

If δ is a direction vector of a particular dependence $S_1 \rightarrow S_2$ in a loop nest and the order of loops in the loop nest is permuted, then the same permutation can be applied to δ to obtain the new direction vector for the conflicting instances of S_1 and S_2.

Legality of Loop Interchange

Direction Matrix

A matrix where each row is the direction vector of a single dependence, i.e.,

each row $\leftrightarrow$ a dependence

each column $\leftrightarrow$ a loop

Legality of Loop Permutation

1. Construct direction matrix
2. Apply the given permutation to the columns of the matrix
3. The permutation is legal *iff* ???

Examples

Legal Case

```
do i=2, N
    do j=2, M-1
        A[i,j] = ... * B[i-1,j-1]
        B[i,j] = ... + A[i,j] + A[i-1,j]
    enddo
enddo
```

$$\begin{bmatrix} = & = \\ < & = \\ < & < \end{bmatrix}$$

Ilegal Case

```
do i=2, N
    do j=2, M-1
        A[i,j] = ... * B[i-1,j-1]
        B[i,j] = ... + A[i,j] + A[i-1,j+1]
    enddo
enddo
```

$$\begin{bmatrix} = & = \\ < & > \\ < & < \end{bmatrix}$$

Profitability of Loop Interchange

Profitability

Profitability is machine-dependent:

1. vector machines
2. parallel machines
3. caches with single outstanding loads
4. caches with multiple outstanding loads

Applying loop interchange

1. Single '$<$' entry: a "serial loop"
   - Move loop outermost for vectorization
   - Move loop innermost for parallelization
2. Multiple '$<$' entries: Outermost one carries dependence
   - Loop carrying the dependence *changes* after permutation!
   - May still benefit by moving carried-dependences to outermost loop

Loop Reversal

Informal Definition

Reverse the order of execution of the iterations of a loop

Example

Scalarize the following Fortran90 statement:

```
A[2:64] = A[1:63] * e
```

```
! Reverse the loop to do this efficiently
do i = 64, 2, -1
    A[i] = A[i-1] * e
enddo
```

Uses

- Scalarize a vector statement (e.g., in Fortran 90) by ensuring that values are read before being written.
- Convert a '$>$' to a '$<$' in a direction vector to enable other transformations, e.g., loop interchange.

Loop Skewing

Informal Definition

Always safe.

Increase dependence distance by n by substituting loop index j with $jj = j + n * i$. E.g., For $n = 1$, we use $jj = j + i$.

Example

```
do i=2,N
    do j=2,N
        A[i,j] = A[i-1,j] + A[i,j-1]
    enddo
enddo
```

```
do i=2,N
    do jj=i+2,i+N
        A[i,jj-i] = A[i-1,jj-i] + A[i,jj-i
    enddo
enddo
```

Uses

- Improve parallelism by converting '$=$' to '$<$' in a direction vector
- Improve vectorization in a similar way
- (Rarely) Could be used to *simplify* index expressions

Unimodular Loop Transformations

These transfomations can be represented by a unimodular transformation matrix T.

- $T\ for\ loop\ interchange = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$
- $T\ for\ loop\ reversal = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$
- $T\ for\ loop\ skewing = \begin{pmatrix} 1 & 0 \\ \alpha & 1 \end{pmatrix}$

1. Any combination of unimodular transformations and inverses of unimodular transformations is also unimodular transformation and can be represented by a unimodular matrix which is the product of the corresponding matrices.
2. A unimodular transformation is legal, if and only if $T\vec{d} \succeq \vec{0}$ for each distance vector $\vec{d}$.

Loop Distribution

Informal Definition

Convert a loop nest containing two or more statements into two or more distinct loop nests so that each statement appears in only a single resulting loop nest.

Example

```
    do i=2,N
S1:     A[i]   = B[i] + C[i]
S2:     D[i]   = A[i] * 2.0
S3:     B[i+1] = A[i] * 3.0
    enddo
```

```
    do i=2,N
S1:     A[i]   = B[i] + C[i]
S3:     B[i+1] = A[i] * 3.0
    enddo
    do i=2,N
S2:     D[i]   = A[i] * 2.0
    enddo
```

Loop Distribution

Motivation

- Convert a loop-carried dependence within a loop into a loop-independent dependence crossing two loops:

```
    do i=2,N
S1:     A[i]   = B[i] + C[i]
S2:     D[i]   = A[i-1] * 2.0
    enddo
```

```
    do i=2,N
S1:     A[i] = B[i] + C[i]
    enddo
    do i=2,N
S2:     D[i] = A[i-1] * 2.0
    enddo
```

- Create perfect loops nests for other transformations like loop interchange

Maximal Loop Distribution: Basic Strategy

Loops Without Branches

1. Identify the SCCs of the data dependence graph

   Idea: Group statements in an SCC in a single loop nest
2. Sort the SCCs using a topological sort on the dependence graph
3. Generate distinct loop nests, one for each SCC, in sorted order

Loops With Branches

Same basic principles as above, but . . .

. . . if $B \xrightarrow{\text{cd}} S2$ and B and $S2$ are in different SCCs:

1. Introduce an "execution variable," `Flag[]`, to record results of B
2. Use `Flag[]` to control execution of $S2$ in later loop nest

Loop Fusion

Informal Definition Merge two or more distinct (perhaps non-adjacent) loops with identical loop bounds into a single loop.

Examples

```
do i=1,N
    A[i] = i*i
enddo
do i=1,N
    B[i] = A[i] + 1
enddo
```

```
do i=1,N
    A[i] = i*i
    B[i] = A[i] + 1
enddo
```

```
do i=1,M
    do j=1,N-1
        A[j,i] = i*i + j*j
    enddo
    do j=1,N
        B[j,i] = A[j,i] + i + j
    enddo
enddo
```

```
! Peel j=N and then fuse:
do i=1,M
    do j=1,N-1
        A[j,i] = i*i + j*j
        B[j,i] = A[j,i] + i + j
    enddo
    j = N
    B[j,i] = A[j,i] + i + j
enddo
```

Loop Fusion

Motivation

1. Increase cache reuse (if same array accessed in two loops)
   $\Leftarrow$ Fundamental optimization for array languages
   Fortran 90, CM-Fortran, HPF, MATLAB, APL
   Example in F90:

```
A[1:M, 1:N] = B[1:M, 1:N] * 2
C[1:M, 1:N] = A[1:M, 1:N] + 1
```

2. Increase granularity of parallelism (work per iteration)
   $\Leftarrow$ Important for shared-memory parallelism
   (parallel loop + barrier model)
   See *Typed Fusion* algorithm in Allen & Kennedy for an example of how fusion can be used to improve shared memory parallelism.

Legality of Loop Fusion

Fusion-Preventing Dependence

A loop-independent dependence from S1 to S2 in different loops is *fusion-preventing* if fusing the two loops causes the dependence to become a loop-carried dependence from S2 to S1.

Legality of Loop Fusion

Two loops can be fused if all 3 conditions hold:

1. Both have identical bounds (*transform loops if needed*)
2. There is no fusion-preventing dependence between them.
3. There is no path of loop-independent dependences between them that contains a loop or statement that is not being fused with them.

Loop Fusion: Illegal Cases

Example 1: Fusion-preventing dependence

```
do i=1,M
    do j=2,N
        A[j,i] = B[j-1,i] * 2
    enddo
    do j=2,N
        B[j,i] = A[j,i] * 3
    enddo
enddo
```

First inner loop can be modified to make fusion legal:

```
do i=1,M
    do j=2,N
        t[j] = B[j-1,i]
    enddo
    do j=2,N
        A[j,i] = t[j] * 2
        B[j,i] = A[j,i] * 3
    enddo
enddo
```

Example 2: Path of loop-independent dependences

```
S1:     A[:, :] = B[:, :] * 2
S2:     C[:, :] = A[:, :] + ...       ! do not want to fuse
S3:     D[:, :] = C[:, :] + A[:, :]   ! want to fuse with S1 but cannot
```

Loop Strip Mining

Informal Definition

Always safe.

Convert a single loop into two nested loops for a specified “block size”

Example

```
do i=1,N
    A[i] = x + B[i] * 2
enddo
```

```
do ii=1,N,B
    do i=ii, min(ii+B-1, N), 1
        A[i] = x + B[i] * 2
    enddo
enddo
```

Uses

- Loop tiling: strip-mine and then interchange
  - multiple uses: for cache; for blocking parallel loops; . . .
- Prefetching: strip-mine by cache line size; prefetch once per outer iteration
- Instruction scheduling: strip-mine and then unroll inner loop

Loop Alignment

Informal Definition

Always safe.

Eliminate a carried dependence by increasing the number of iterations and executing statements on different subsets of the iterations.
(*Alternative to loop distribution*)

Example

```
do i=2,N
    A[i] = B[i] + C[i]
    D[i] = A[i-1] * 2.0
enddo
```

```
i = 1
D[i+1] = A[i] * 2.0
do i=2,N-1
    A[i]   = B[i] + C[i]
    D[i+1] = A[i] * 2.0   ! note subscripts
enddo
i = N
A[i] = B[i] + C[i]
```

Uses

- Improve parallelism by eliminating carried dependences

Scalar Replacement

Informal Definition

Replace an array reference with a scalar temporary.
Note: Use dependences to locate consistent re-use patterns

Example

```
do i = 1, n
    do j = 2, n
        x(j,i) = a(i) + x(j-1,i)
                      + b(j,i)
    enddo
enddo
```

```
do i = 1, n
    t1 = a(i)
    t2 = x(1,i)
    do j = 2, n
        x(j,i) = t1 + t2 + b(j,i)
        t2     = x(j,i)
    enddo
enddo
```

Uses

- decrease number of loads and stores
- enable allocator to keep (some) array refs in registers

$\Rightarrow$ often see improvements by factors of $2\times$ to $3\times$

Unroll and Jam

Balance between Memory Accesses and Computation

machine balance: $\beta_M = \frac{\text{fetches/cycle}}{\text{flops/cycle}}$

loop balance: $\beta_L = \frac{\text{fetches/iteration}}{\text{flops/iteration}}$

We want loops to be balanced or slightly compute-bound

- if $\beta_L > \beta_M$, loop is *memory-bound*
- if $\beta_L < \beta_M$, loop is *compute-bound*

Balance is becoming more important

- memory latencies are increasing
- flops/cycle is increasing

Unroll and Jam: Improving Balance

Informal Definition

1. unroll the outer loop by k
2. fuse the resulting k inner loops into a single loop

Simple example:

```
do i = 1, n
    do j = 1, n
        a(i) = a(i) + b(j)
    enddo
enddo
```

Unroll-and-jam ($k = 2$):

```
do i = 1, n-1, 2
    do j = 1, n
        a(i)   = a(i)   + b(j)
        a(i+1) = a(i+1) + b(j)
    enddo
enddo
```

Scalar replacement works well for references to a(i) but does not extract any reuse for b(j).

$$\Longrightarrow \qquad \beta_L = \frac{\text{1 fetch}}{\text{1 flop}}$$

Unroll and Jam: Improving Balance (cont'd)

Unroll-and-jam Improves Balance:

```
do i = 1, n, 2
 t0 = a(i+0)
 t1 = a(i+1)
 do j = 1, n
     t3 = b(j)
     t0 = t0 + t3
     t1 = t1 + t3
 enddo
 a(i+0) = t0
 a(i+1) = t1
enddo
```

- $\beta_L = \frac{\text{1 fetch}}{\text{2 flops}}$
- Uses one extra register for the inner loop.
- Allows more efficient pipelining (if $+$ take
  $>= 2$ cycles)