Advanced Computer Architecture

Chapter 5:
Compiler issues: dependence analysis, vectorisation, automatic parallelisation

Lecturer: *Paul H J Kelly*

Autumn 2001

Department of Computing
Imperial College

Room 423
email: phjk@doc.ic.ac.uk

Course web site:
http://www.doc.ic.ac.uk/~phjk/AdvancedComputerArchitecture.html

CHAPTER 5

Introduction

In this segment of the course we consider compilation issues for loops involving arrays:

- How execution order of a loop is constrained,
- How a compiler can extract dependence information, and
- How this can be used to optimise a program.

Understanding and transforming exec^n order can exploit architectural features:

- Pipelined, superscalar and VLIW processors
- Systems which rely heavily on caches.
- CPUs with special instructions for vectors.
- Multiprocessors.

5.1 Background reading

The material for this part of the course is introduced only very briefly in Hennessy and Patterson. A good textbook which covers it properly is

> Michael Wolfe, "High Performance Compilers for Parallel Computing" [4].

Much of the presentation is taken from the following research paper:

> U. Banerjee, "Unimodular transformations of double loops" [2].

Banerjee's paper gives a simplified account of the theory in the context only of perfect doubly-nested loops with well-known dependences.

Background reading, cont'd

Monica Lam's group at Stanford have used these ideas to build a public-domain prototype compiler system "SUIF". The following paper describes how SUIF uses unimodular transformations to optimise locality:

> M.E. Wolf and M.S. Lam, "A data locality optimizing algorithm" [3].

The Stanford work overcomes many of the restrictions of Banerjee's paper, but lies beyond the scope of this course.

Restructuring

- Here we consider a special kind of optimisation, which is currently performed only by specialist compilers — "restructuring compilers".
- Conventional optimisations (see the Dragon [1] book) must also be performed.

The difference is this:

- ⋆ Conventional optimisations reduce the amount of work the computer has to do at run-time.
- ⋆ Restructuring aims to do the work in an order which suits the target architecture better.

Outline

- **Motivation:** exec^{n} order really matters, even on fairly "ordinary" architectures.
- **Dependence analysis:** how compiler can analyse array subscript exp^{n}s in order to determine whether restructuring would be valid - e.g.
  - Loop interchange
  - Loop fusion
  - Loop reversal
  - Loop parallelisation (for parallel or pipelined machines)
  - Loop vectorisation (machines with vector registers and vector instructions)
- **Loop transformation using unimodular matrices:** given the dependence structure, how does the compiler efficiently find the optimum loop transformation?

5.2 Motivation: an example

- We will begin by looking at matrix multiplication.
- We will study the performance of a 512×512 double-precision floating point matrix multiply.
- We will investigate the performance of various versions in order to determine what transformations a compiler should be applying.

Initial version

```
/*
 * mm: Multiply A by B leaving the
 *     result in C.
 *     The result matrix is assumed
 *     to be initialised to zero.
 */
void mm1(A,B,C)
     double A[512][512],
            B[512][512],
            C[512][512];
{
  int i, j, k;

  for (i = 0; i < 512; i++)
    for (j = 0; j < 512; j++)
      for (k = 0; k < 512; k++)
        C[i][j] += A[i][k] * B[k][j];
}
```


Architectural details

The experiments were performed on an IBM Thinkpad T21 laptop with 128MB RAM:

- This machine has an 800MHz Intel Pentium III processor pipelined RISC CPU.
- 20 Entry RS, 40 Entry ROB
- Pipeline Depth: 12 (in-order) plus 2 (out-of-order) stages
- "up to 5 μOPs/Cycle"

The compiler used was Microsoft Visual Studio 6.0

Memory system

- L1 Instruction cache: 16 KB, 4-Way, 32 Byte/Line, LRU
- L1 Data cache: 16 KB, 4-way, 32 byte/line, non-blocking, dual-ported, write allocate, LRU
- L2 unified cache: 256 KB, 8-Way, 32 Byte/Line, non-blocking
- Note that the matrix occupies $512^2 \times 8 \approx 2$MB.
- Each row of the matrix occupies $512 \times 8 \approx 4$KB.

Performance

- The initial version runs in 10.3 seconds.
- The matrix multiplication takes 512^3 steps, each involving two floating-point operations, an add and a multiply, i.e. 268×10^6.
- This loop achieves a computation rate of 268/10.3=26.1 MFLOPs.
- That is, one floating-point operation completed every 30 clock cycles

  How are we going to get value for money?

Interchange loops

```
for (i = 0; i < 512; i++)
  for (k = 0; k < 512; k++){
    r = A[i][k];
    for (j = 0; j < 512; j++)
      C[i][j] += r * B[k][j];
  }
```

- 5.2 seconds (51.6 MFLOPS).
- Why is this such a good idea?
- How might a compiler perform this transformation?
- Can we do better still?

Optimising memory accesses

- What is the data cache behaviour of our program?
- How can it be improved?
- Why do cache hits occur?
  - Spatial locality
  - Temporal locality
- Why do cache misses occur?
  - Compulsory (never seen before)
  - Capacity (cache not big enough)
  - Conflicts (cache not associative enough)

How can we arrange for the data to be in the cache already?

Blocking (a.k.a. "tiling")

Idea: reorder exec^n of loop nest so data isn't evicted from cache before it's needed again.

Blocking is a combination of two transformations: "strip mining", followed by interchange; we start with

```
for (i = 0; i < 512; i++)
  for (k = 0; k < 512; k++){
    r = A[i][k];
    for (j = 0; j < 512; j++)
      C[i][j] += r * B[k][j]; }
```

Strip mine the `k` and `j` loops:

```
for (i = 0; i < 512; i++)
 for (kk = 0; kk < 512; kk += BLKSZ)
   for (k = kk; k < min(kk+BLKSZ,512); k++){
    r = A[i][k];
    for (jj = 0; jj < 512; jj += BLKSZ)
     for (j = jj; j < min(jj+BLKSZ, 512); j++)
       C[i][j] += r * B[k][j];
    }
```


Now interchange so blocked loops are outermost:

```
for (kk = 0; kk < 512; kk += BLKSZ)
 for (jj = 0; jj < 512; jj += BLKSZ)
  for (i = 0; i < 512; i++)
   for (k = kk; k < min(kk+BLKSZ,512); k++){
    r = A[i][k];
     for (j = jj; j < min(jj+BLKSZ, 512); j++)
      C[i][j] += r * B[k][j];
    }
```

- The inner i,k,j loops perform a multiplication of a pair of $BLKSZ \times BLKSZ$ partial matrices.
- $BLKSZ$ is chosen so that a $BLKSZ \times BLKSZ$ submatrix of `B` and a row of length $BLKSZ$ of `C` can fit in the cache.
- What is the right value for $BLKSZ$?

Performance of blocked version

Blocking factor	Execn time	MFLOPS
8	3.815	70.4
16	2.784	96.4
32	2.283	117.6
40	2.193	122.4
48	2.253	119.1
56	2.473	108.5
64	3.404	78.9
72	5.608	47.9
80	5.578	48.1
88	5.808	46.2
96	5.928	45.3
104	6.309	42.5
112	5.778	46.5

Reducing overheads of blocking...

The min operators in the inner loop bounds are likely to be a performance hit; if we choose a good blocking factor which divides the problem size exactly...

Blocking factor	Execn time	MFLOPS
32	2.013	133.4
64	2.915	92.1

Summary

- Original version: 10.3 seconds (26.1 MFLOPS).
- That was using a "good" optimising compiler!
- Factor of five performance improvement.
- No reduction in amount of arithmetic performed.

(The CodePlay VectorC compiler generates slightly better code for this inner, blocked loop, giving 155.0 MFLOPS)

Moral

- "Sophisticated" architectures are commonplace.
- Conventional optimisation must be augmented with techniques to deal with complex architectures. Examples include
  - Instruction scheduling.
  - Software pipelining.
  - Loop interchange.
  - Blocking.
- The compiler writer must solve two distinct problems:
  - Analysis: under what precise conditions can each transformation be applied?
  - Transformation: given the semantic constraints, what is the best code to generate?

5.3 Dependence

Define:

- $IN(S)$: set of memory loc^{n}s which *might* be *read* by some exec^{n} of statement S
- $OUT(S)$: set of memory loc^{n}s which *might* be *written* by some exec^{n} of statement S

Reordering is constrained by dependences;
There are four types:

- Data ("true") dependence: $S_1 \; \delta \; S_2$
  $OUT(S_1) \cap IN(S_2)$
- Anti dependence: $S_1 \; \overline{\delta} \; S_2$
  $IN(S_1) \cap OUT(S_2)$
- Output dependence: $S_1 \; \delta^o \; S_2$
  $OUT(S_1) \cap OUT(S_2)$
- Control dependence: $S_1 \; \delta^c \; S_2$

These are static analogues of dynamic RAW, WAR, WAW and control hazards.

Loop-carried dependences

Consider:

S_1: `A[0] := 0`
`for I = 1 to 5`
S_2: `A[I] := A[I-1] + B[I]`

What does this loop do?

Unroll:

S_1: $\boxed{\texttt{A[0]}}$ `:= 0`
S_2^1: $\boxed{\texttt{A[1]}}$ `:=` $\boxed{\texttt{A[0]}}$ `+ B[1]`
S_2^2: `A[2] :=` $\boxed{\texttt{A[1]}}$ `+ B[2]`
S_2^3: `A[3] := A[2] + B[3]`
S_2^4: `A[4] := A[3] + B[4]`
S_2^5: `A[5] := A[4] + B[5]`

- In this case, there is a data dependence $S_2^{i-1} \; \delta \; S_2^i$
- This is a loop-carried dependence - the dependence spans a loop iteration
- This loop is inherently sequential

Loop-carried dependences...cont'd

What is a loop-carried dependence?

- Consider two iterations I_1 and I_2
- A dependence occurs between two statements S_p and S_q (not necessarily distinct), when an assignment in $S_p^{I_1}$ refers to the same location as a use in $S_q^{I_2}$
- In the example,

  S_1: `A[0] := 0`
  `for I = 1 to 5`
  S_2: `A[I] := A[I-1] + B[I]`

  - The assignment is "`A[`I_1`] := ...`"
  - The use is "`... := A[`I_2`-1] ...`"
  - These refer to the same location when $I_1 = I_2 - 1$
  - Thus $I_1 < I_2$, ie the assignment is in an earlier iteration
- Notation: $S_2 \; \delta_< \; S_2$

Definition: The dependence equation

- A dependence occurs between two statements S_p and S_q (not necessarily distinct), when there exists a pair of loop iterations I_1 and I_2, such that a memory reference in $S_p^{I_1}$ may refer to the same location as memory reference in $S_q^{I_2}$.
- This might occur if S_p and S_q refer to some common array `A`
- Suppose S_p refers to $\mathtt{A}[\phi_p(I)]$
- Suppose S_q refers to $\mathtt{A}[\phi_q(I)]$
- A dependence of some kind occurs between S_p and S_q if there exists a solution to the equation

  $$\phi_p(I_1) = \phi_q(I_2)$$

  for integer values of I_1 and I_2 lying within the loop bounds

Types of dependence

- If a solution to the dependence equation exists, a dependence of some kind occurs
- The dependence type depends on what solutions exist
- The solutions consist of a set of pairs (I_1, I_2)
- We would appear to have a *data* dependence if

  $$\mathtt{A}[\phi_p(I)] \in OUT(S_p)$$

  and

  $$\mathtt{A}[\phi_q(I)] \in IN(S_q)$$

- But we only really have a data dependence if the assignments *precede* the uses, ie
  - $S_p\ \delta_<\ S_q$ if, for each solution pair (I_1, I_2), $I_1 < I_2$

- If the *uses* precede the *assignments*, we actually have an *anti*-dependence, ie
  - $S_p\ \overline{\delta}_<\ S_q$ if, for each solution pair (I_1, I_2), $I_1 > I_2$
- If there are some solution pairs (I_1, I_2) with $I_1 < I_2$ and some with $I_1 > I_2$, we write $S_p\ \delta_*\ S_q$
- If, for all solution pairs (I_1, I_2), $I_1 = I_2$, there are dependences *within* an iteration of the loop, but there are no loop-carried dependences.

5.4 Dependence distance

In many common examples, the set of solution pairs is characterised easily:

Definition: dependence distance

- If, for all solution pairs (I_1, I_2),

  $$I_1 = I_2 - k$$

  then the dependence distance is k

For example in the loop we considered earlier,

```
S1 : A[0] := 0
     for I = 1 to 5
S2 :   A[I] := A[I-1] + B[I]
```

We find that $S_2\ \delta_<\ S_2$ with dependence distance 1.

((of course there are many cases where the difference is not constant and so the dependence cannot be summarised this way)).

5.5 Reuse distance

When optimising for cache performance, it is sometimes useful to consider the re-use relationship,

$$IN(S_1) \cap IN(S_2)$$

- Here there is no dependence - it doesn't matter which read occurs first
- Nonetheless, cache performance can be improved by minimising the *reuse distance*
- The reuse distance is calculated essentially the same way

5.6 Nested loops

We begin by considering a very simple loop with a very common dependence pattern. This example is also used in Banerjee's paper:

for I_1 = 0 to 3 do
  for I_2 = 0 to 3 do
$S:$ A[I_1,I_2] := A[$I_1 - 1$,I_2] + A[I_1,$I_2 - 1$]

Dependence structure?

Consider the dependence equations:

for I_1 = 0 to 3 do
  for I_2 = 0 to 3 do
$S:$ A[I_1,I_2] := A[$I_1 - 1$,I_2] + A[I_1,$I_2 - 1$]

There are two potential dependences arising from the three references to A.

Therefore two systems of dependence equations to solve:

1. Between A[I_1^1,I_2^1] and A[$I_1^2 - 1$,I_2^2]:

$$\begin{cases} I_1^1 = I_1^2 - 1 \\ I_2^1 = I_2^2 \end{cases}$$

2. Between A[I_1^1,I_2^1] and A[I_1^2,$I_2^2 - 1$]:

$$\begin{cases} I_1^1 = I_1^2 \\ I_2^1 = I_2^2 - 1 \end{cases}$$

((strictly we should also consider output dependences between A[I_1^1,I_2^1] and A[I_1^2,I_2^2], but this is obviously absent)).

Equation 1 has solutions for

$$\begin{aligned} ((I_1^1, I_2^1), (I_1^2, I_2^2)) &= ((0,0),(1,0)), \\ &= ((1,0),(2,0)), \\ &= ((2,0),(3,0)), \\ &= ((0,1),(1,1)), \\ &= ((1,1),(2,1)), \\ &= ((2,1),(3,1)), \\ &\vdots \\ &= ((0,3),(1,3)), \\ &= ((1,3),(2,3)), \\ &= ((2,3),(3,3)) \end{aligned}$$

We can summarise this by saying that the *dependence distance vector* for this dependence is

$$(I_1^2 - I_1^1, I_2^2 - I_2^1)$$

that is, $(1, 0)$.

This corresponds to a *dependence direction vector* $(<, .)$: there is a loop-carried data dependence in the outer loop.

Equation 2 has solutions for

$$
\begin{aligned}
((I_1^1, I_2^1), (I_1^2, I_2^2)) &= ((0,0),(0,1)),\\
&= ((0,1),(0,2)),\\
&= ((0,2),(0,3)),\\
&= ((1,0),(1,1)),\\
&= ((1,1),(1,2)),\\
&= ((1,2),(1,3)),\\
&\quad \ldots\\
&= ((3,0),(3,1)),\\
&= ((3,1),(3,2)),\\
&= ((3,2),(3,3))
\end{aligned}
$$

The distance vector is

$$(I_1^2 - I_1^1, I_2^2 - I_2^1) = (0,1)$$

This corresponds to a dependence direction vector $(., <)$: there is also a loop-carried data dependence in the inner loop.

((Note that it is possible for a dependence to exist without having a consistent dependence distance)).

The same loop:

```
for I₁ = 0 to 3 do
  for I₂ = 0 to 3 do
S :   A[I₁,I₂] := A[I₁ − 1,I₂] + A[I₁,I₂ − 1]
```

For humans the easy way to understand this loop nest is to draw the iteration space showing the iteration-to-iteration dependences:

$$
\begin{array}{ccccccc}
S^{00} & \xrightarrow[\delta]{} & S^{01} & \xrightarrow[\delta]{} & S^{02} & \xrightarrow[\delta]{} & S^{03}\\
\delta\downarrow & & \delta\downarrow & & \delta\downarrow & & \delta\downarrow\\
S^{10} & \xrightarrow[\delta]{} & S^{11} & \xrightarrow[\delta]{} & S^{12} & \xrightarrow[\delta]{} & S^{13}\\
\delta\downarrow & & \delta\downarrow & & \delta\downarrow & & \delta\downarrow\\
S^{20} & \xrightarrow[\delta]{} & S^{21} & \xrightarrow[\delta]{} & S^{22} & \xrightarrow[\delta]{} & S^{23}\\
\delta\downarrow & & \delta\downarrow & & \delta\downarrow & & \delta\downarrow\\
S^{30} & \xrightarrow[\delta]{} & S^{31} & \xrightarrow[\delta]{} & S^{32} & \xrightarrow[\delta]{} & S^{33}
\end{array}
$$

The diagram shows an arrow for each solution of each dependence equation.

Note that:

- This loop is *interchangeable*: the top-to-bottom, left-to-right execution order is also valid since all dependence constraints (as shown by the arrows) are still satisfied.
- The inner loop is not vectorisable since there is a dependence chain linking successive iterations.
  - (to use a vector instruction, need to be able to operate on each element of the vector in parallel)
- Similarly, the outer loop is not parallel
- Interchanging the loop does not improve vectorisability or parallelisability

5.7 Interchange: counterexample

$$
\begin{tikzcd}
S^{00} & S^{01} \arrow[dl] & S^{02} \arrow[dl] & S^{03} \arrow[dl]\\
S^{10} & S^{11} \arrow[dl] & S^{12} \arrow[dl] & S^{13} \arrow[dl]\\
S^{20} & S^{21} \arrow[dl] & S^{22} \arrow[dl] & S^{23} \arrow[dl]\\
S^{30} & S^{31} & S^{32} & S^{33}
\end{tikzcd}
$$

```
for I₁ = 0 to 3 do
  for I₂ = 0 to 3 do
S :   A[I₁,I₂] := A[I₁ + 1,I₂ − 1] + ...
```

5.8 Interchange: condition

- This loop is *interchangeable*: the top-to-bottom, left-to-right execution order is also valid since all dependence constraints (as shown by the arrows) are still satisfied.
- How can you tell whether a loop can be interchanged?
- Look at it's dependence direction vectors:
  - If any dependence direction vector has the form $(<, >)$, interchange is *invalid*
  - Because the arrows would be traversed backwards
- All other dependence directions are OK.

5.9 Skewing

Consider the following loop:

```
for k1 := 0 to 3 do
  for k2 := k1 to k1+3 do
S : A[k1,k2-k1] := A[k1-1,k2-k1]+A[k1,k2-k1-1]
```

- The inner loop's control variable runs from $\mathtt{k_1}$ to $\mathtt{k_1}$+3.
- The iteration space of this loop has 4^2 iterations just like the original loop.
- If we draw the iteration space with each iteration S^{K_1,K_2} at coordinate position (K_1, K_2), it is skewed to form a lozenge shape:

$$\begin{array}{ccccccc}
S^{00} & S^{01} & S^{02} & S^{03} & & & \\
 & S^{11} & S^{12} & S^{13} & S^{14} & & \\
 & & S^{22} & S^{23} & S^{24} & S^{25} & \\
 & & & S^{33} & S^{34} & S^{35} & S^{36}
\end{array}$$

- This loop performs the same computation as the original.

- To see that this loop performs the same computation, lets work out its dependence structure.
- First label each iteration with the element of `A` to which it assigns:

S^{00} A00	S^{01} A01	S^{02} A02	S^{03} A03			
	S^{11} A10	S^{12} A11	S^{13} A12	S^{14} A13		
		S^{22} A20	S^{23} A21	S^{24} A22	S^{25} A23	
			S^{33} A30	S^{34} A31	S^{35} A32	S^{36} A33

- The loop body is

```
A[k1,k2-k1] := A[k1-1,k2-k1]+A[k1,k2-k1-1]
```

- E.g. iteration S_{23} does:

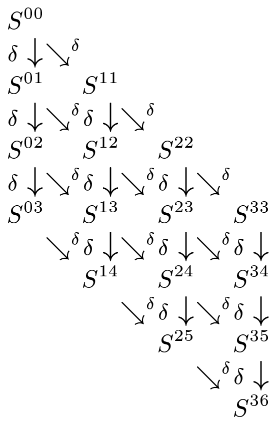
```
A[2,1] := A[1,1]+A[2,0]
```


Thus the dependence structure of the skewed loop is shown by marking the iteration space with all the dependences:

$$\begin{tikzcd}
S^{00} \arrow[r, "\delta"'] \arrow[dr, "\delta"] & S^{01} \arrow[r, "\delta"'] \arrow[dr, "\delta"] & S^{02} \arrow[r, "\delta"'] \arrow[dr, "\delta"] & S^{03} \arrow[dr, "\delta"] & & & \\
 & S^{11} \arrow[r, "\delta"'] \arrow[dr, "\delta"] & S^{12} \arrow[r, "\delta"'] \arrow[dr, "\delta"] & S^{13} \arrow[r, "\delta"'] \arrow[dr, "\delta"] & S^{14} \arrow[dr, "\delta"] & & \\
 & & S^{22} \arrow[r, "\delta"'] \arrow[dr, "\delta"] & S^{23} \arrow[r, "\delta"'] \arrow[dr, "\delta"] & S^{24} \arrow[r, "\delta"'] \arrow[dr, "\delta"] & S^{25} \arrow[dr, "\delta"] & \\
 & & & S^{33} \arrow[r, "\delta"'] & S^{34} \arrow[r, "\delta"'] & S^{35} \arrow[r, "\delta"'] & S^{36}
\end{tikzcd}$$

- Can this loop nest be vectorised?
- Can this loop nest be interchanged?

After interchange




```
for k2 := ?  to ?  do
  for k1 := ?  to ?  do
S : A[k1,k2-k1] := A[k1-1,k2-k1]+A[k1,k2-k1-1]
```

- The inner loop is now vectorisable.
- The skewed iteration space has N rows and $2N-1$ columns, but still only N^2 actual statement instances.
- What are the appropriate loop bounds?

For $N_1 = 4$, $N_2 = 5$

S^{00}	.	.	.	.
S^{01}	S^{11}	.	.	.
S^{02}	S^{12}	S^{22}	.	.
S^{03}	S^{13}	S^{23}	S^{33}	.
.	S^{14}	S^{24}	S^{34}	S^{44}
.	.	S^{25}	S^{35}	S^{45}
.	.	.	S^{36}	S^{46}
.	.	.	.	S^{47}

Interchanged, skewed loop:

```
for k2 := 0 to 8 do
  for k1 := max(0,K2-3) to min(K2,4) do
S : A[k1,k2-k1] := A[k1-1,k2-k1]+A[k1,k2-k1-1]
```

That is,

```
for k2 := 0 to 2N2-2 do
  for k1 := max(0,K2-N2+2) to min(K2,N1) do
S : A[k1,k2-k1] := A[k1-1,k2-k1]+A[k1,k2-k1-1]
```


Summary

- Original loop interchangeable but not vectorisable.
- We skewed inner loop by outer loop by factor 1.
- Still not vectorisable, but interchangeable.
- Interchanged, skewed loop *is* vectorisable.
- Bounds of new loop not simple!

Questions about skewing:

— Is skewing ever invalid?

— Does skewing affect interchangeability?

— Does skewing affect dependence distances?

— Can you predict the usefulness of skewing?

5.10 Matrix representation of loop transformations

To skew the inner loop by the outer loop by factor 1 we adjust the loop bounds, and replace I_1 by K_1, and I_2 by $K_2 - K_1$. That is,

$$(K_1, K_2) = (I_1, I_2) \cdot \mathbf{U}$$

where $\mathbf{U}$ is a 2×2 matrix

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$

That is,

$$(K_1, K_2) = (I_1, I_2) \cdot \mathbf{U} = (I_1, I_2 + I_1)$$

The inverse gets us back again:

$$(I_1, I_2) = (K_1, K_2) \cdot \mathbf{U}^{-1} = (K_1, K_2 - K_1)$$

Matrix **U** maps each statement instance $S^{I_1 I_2}$ to its position in the new iteration space, $S^{K_1 K_2}$:

Original iteration space:

I_1 \ I_2:	0	1	2	3
0	S^{00}	S^{01}	S^{02}	S^{03}
1	S^{10}	S^{11}	S^{12}	S^{13}
2	S^{20}	S^{21}	S^{22}	S^{23}
3	S^{30}	S^{31}	S^{32}	S^{33}

Transformed iteration space:

K_1 \ K_2:	0	1	2	3	4	5	6
0	S^{00}	S^{01}	S^{02}	S^{03}			
1		S^{11}	S^{12}	S^{13}	S^{14}		
2			S^{22}	S^{23}	S^{24}	S^{25}	
3				S^{33}	S^{34}	S^{35}	S^{36}

The dependences are subject to the same transformation.

Recall that:

- There is a dependence between two iterations (I_1^1, I_2^1) and (I_1^2, I_2^2) if there is a memory location which is assigned to in iteration (I_1^1, I_2^1), and read in iteration (I_1^2, I_2^2).
  ((unless there is an intervening assignment))
- If (I_1^1, I_2^1) precedes (I_1^2, I_2^2) it is a *data*-dependence.
- If (I_1^2, I_2^2) precedes (I_1^1, I_2^1) it is a *anti*-dependence.
- If the location is assigned to in both iterations, it is an *output*-dependence.
- The dependence distance vector (D_1, D_2) is $(I_1^2 - I_1^1, I_2^2 - I_2^1)$.

- Iterations $(I_1^1, I_2^1) \cdot \mathbf{U}$ and $(I_1^2, I_2^2) \cdot \mathbf{U}$ will also read and write the same location.
- The transformation **U** is *valid* iff

  $$(I_1^1, I_2^1) \cdot \mathbf{U} \text{ precedes } (I_1^2, I_2^2) \cdot \mathbf{U}$$

  whenever there is a dependence between (I_1^1, I_2^1) and (I_1^2, I_2^2).
- In the transformed loop the dependence distance vector is also transformed, to

  $$(D_1, D_2) \cdot \mathbf{U}$$

Example: loop given earlier

Before transformation we had two dependences:

1. Distance: $(1, 0)$, direction: $(<, .)$
2. Distance: $(0, 1)$, direction: $(., <)$

After transformation by matrix $\mathbf{U} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$

(i.e. skewing of inner loop by outer) we get:

1. Distance: $(1, 1)$, direction: $(<, <)$
2. Distance: $(0, 1)$, direction: $(., <)$

We can also represent loop interchange by a matrix transformation.

After transforming the skewed loop by matrix $\mathbf{V} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ (i.e. loop interchange) we get:

1. Distance: $(1,1)$, direction: $(<,<)$
2. Distance: $(1,0)$, direction: $(<,.)$

The transformed iteration space is the transpose of the skewed iteration space:

$$\begin{array}{llll}
S^{00} & & & \\
S^{10} & S^{11} & & \\
S^{20} & S^{21} & S^{22} & \\
S^{30} & S^{31} & S^{32} & S^{33} \\
 & S^{41} & S^{42} & S^{43} \\
 & & S^{52} & S^{53} \\
 & & & S^{63}
\end{array}$$

Summary

- $(I_1, I_2) \cdot \mathbf{U}$ maps each statement instance (I_1, I_2) to its new position (K_1, K_2) in the transformed loop's exec^{n} sequence
- $(D_1, D_2) \cdot \mathbf{U}$ gives new dependence distance vector, giving test for validity
- Captures skewing, interchange and reversal
- Compose transformations by matrix multiplication $\mathbf{U_1} \cdot \mathbf{U_2}$
- Resulting loop's bounds may be a little tricky

⋆ Efficient algorithms exist [2] to maximise parallelism by skewing and loop interchanging

⋆ Efficient algorithms exist to optimise cache performance by finding the combination of blocking, block size, interchange and skewing which leads to the best reuse [3]

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison Wesley, 1986.

[2] U. Banerjee. Unimodular transformations of double loops. In *Proceedings of the Third Workshop on Programming Languages and Compilers for Parallel Computing, Irvine, CA.* Pitman/MIT Press, 1990.

[3] M.E. Wolf and M.S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, volume 26, pages 30–44, Toronto, Ontario, Canada, June 1991.

[4] Michael Wolfe. *High Performance Compilers for Parallel Computing.* Addison Wesley, 1996.