Lecture 7: Loop Transformations

Spring 2010

Maria J. Garzaran

Data Dependences

- The correctness of many many loop transformations can be decided using dependences.

- Still a good introduction to the notion of dependence and its applications can be found in *D. Kuck, R. Kuhn, D. Padua, B. Leasure, M. Wolfe: Dependence Graphs and Compiler Optimizations. POPL 1981.*

- *For a longer discussion see:U. Banerjee. Dependence Analysis for Supercomputing. Kluwer Academic Publish- ers, Norwell, Mass., 1988*

Compiler Optimizations

1) Advanced Compiler Optimizations for Supercomputers, by David Padua and Michael Wolfe in Communications of the ACM, December 1986, Volume 29, Number 12.

2) Compiler Transformations for High-Performance Computing, by David Bacon, Susan Graham and Oliver Sharp, in ACM Computing Surveys, Vol. 26, No. 4, December 1994.

Dependences

Flow Dependence (True Dependence)

```
S1    X=A+B
S2    C=X+1
```

$$\begin{tikzcd} S1 \arrow[d] \\ S2 \end{tikzcd}$$

Anti Dependence

```
S1    A=X+B
S2    X=C+D
```

$$\begin{tikzcd} S1 \arrow[d, "-" marking] \\ S2 \end{tikzcd}$$

Output Dependence

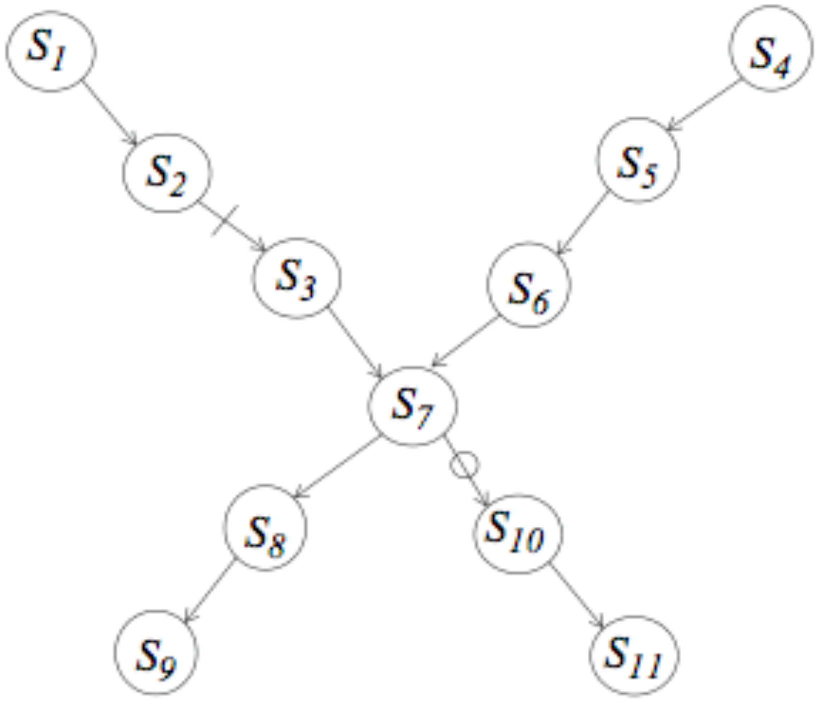
```
S1    X=A+B
      . . .
S2    X=C+D
```

$$\begin{tikzcd} S1 \arrow[d, "\circ" marking] \\ S2 \end{tikzcd}$$

The notion of dependence applies to **sequential programs**.

Transformations are sequential to sequential or sequential to parallel.

Dependences and ILP



S_1; S_2; S_3 can execute in parallel with S_4; S_5; S_6

S_8; S_9 " " " " " S_{10}; S_{11}

Removal of Output and Antidependences

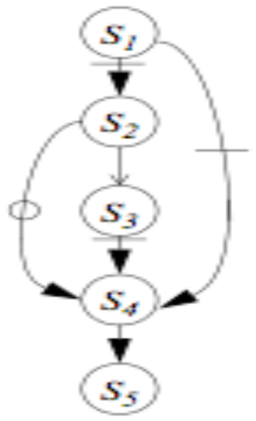
- Variable Renaming
- Scalar Expansion
- Node Splitting

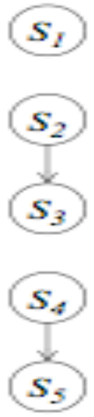
Renaming

```
S1      A=X+B
S2      X=Y+1
S3      C=X+B
S4      X=Z+B
S5      D=X+1
```

renaming

```
S1      A=X+B
S2      X1=Y+1
S3      C=X1+B
S4      X2=Z+B
S5      D=X2+1
```





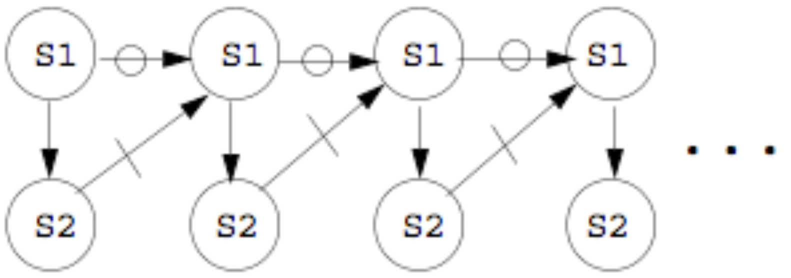
Data Dependences

```
for (i=0; i<4; i++) {
  a[i]= b[i];
  c[i] = c[i+1] + a[i];
}
```

- Data dependences between statement instances that belong to the same loop iteration are called loop-independent.
- Data dependences between statements instances that belong to different loop iterations are called loop-carried.

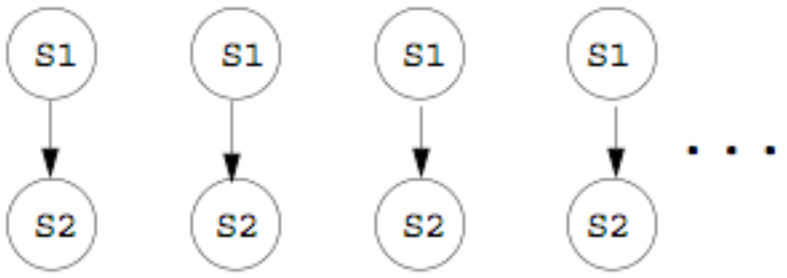
Scalar expansion

```
      DO    I=1,N
S1:         A=B(I)+1
S2:         C(I)=A+D(I)
      END DO
```



scalar expansion

```
      DO    I=1,N
S1:         A1(I)=B(I)+1
S2:         C(I)=A1(I)+D(I)
      END DO
      A=A1(N)
```

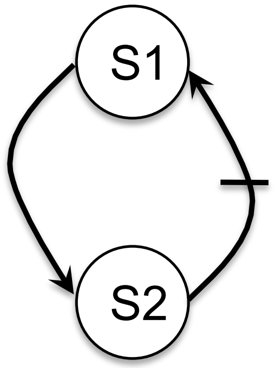


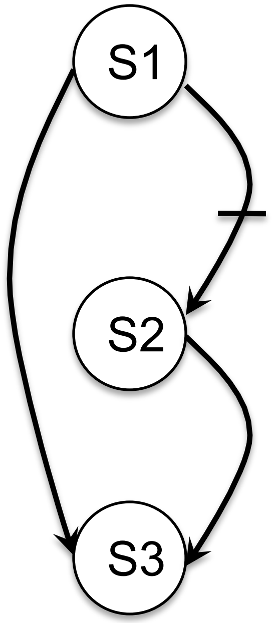
Node Splitting

- Some loops contain data-dependence cycles that can be easily eliminated by copying data.

```
for (i=0; i<N; i++){
S1:   a[i]= b[i] + c[i];
S2:   d[i] = (a[i] + a[i+1])/2;
}
```

```
for (i=0; i<N; i++){
  temp[i] = a[i+1]
  a[i] = b[i+ c[i];
  d[i] = (a[i] + temp[i])/2;
}
```



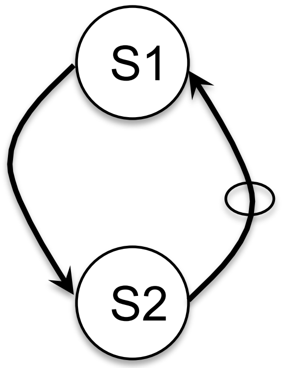


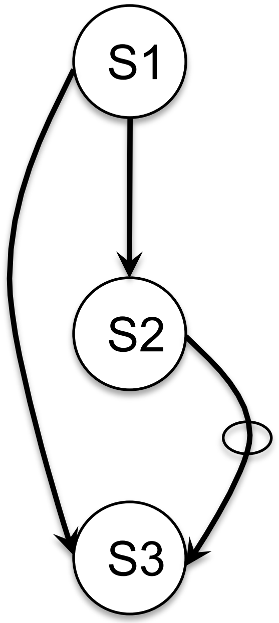
Node Splitting

```
for (i=0; i<N; i++){
S1:   a[i]= b[i] + c[i];
S2:   a[i+1] = a[i] + 2 * d[i];
}
```

```
for (i=0; i<N; i++){
S1:   temp[i] = b[i] + c[i];
S2:   a[i+1]= temp[i]+ 2 * d[i];;
S3:   a[i]= temp[i];
}
```

Removal of output dependences in data-dependence cycles





Loop Optimizations

- Loop Distribution or loop fission
- Loop Fusion
- Loop Peeling
- Loop Unrolling
- Unroll and Jam
- Loop Interchaging
- Loop reversal
- Strip Mining
- Loop Tiling
- Software Pipelining

Loop Distribution

- It is also called loop fission.
- Divides loop control over different statement in the loop body.

```
for (i=1; i<100; i++) {
  a[i]= b[i];
  c[i] = c[i-1] + 1;
}
```

```
for (i=1; i<100; i++)
  a[i]= b[i];

for (i=1; i<100; i++)
  c[i] = c[i-1] + 1;
```

Loop Distribution

- It is valid if no loop-carried data dependences exist that are lexically backward, that is, going from one statement instance to an instance of a statement that appears earlier in the loop body.

```
for (i=0; i<100; i++) {
  a[i]= b[i] + c[i]
  d[i] = a[i+1];
}
```

Loop Distribution

- This transformation is useful for
  - Isolating data dependences cycles in preparation for loop vectorization
  - Enabling other transformations, such as loop interchanging
  - Improving locality by reducing the total amount of data that is referenced during complete execution of each loop.
  - Separate different data streams in a loop to improve hardware prefetch

Loop Distribution

```
for (i=0; i<N; i++) {
  buff1[i]= 0;
  buff2[i] = 0;


  ….
  buffn[i]=0;
}
```

```
for (i=0; i<N; i++) {
  buff1[i] = 0;
  ….
  buff4[i] = 0;
}

for (i=0; i<N; i++) {
  buff5[i] = 0;
  ….
  buff8[i] = 0;
}
```

Loop Distribution

- Plot from the book

Loop Fusion

- This transformation merges adjacent loops with identical control into one loop.

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

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

Loop Fusion

- This transformation is valid if the fusion does not introduce any lexically backward data dependence.

```
for (i=2; i<N; i++)
  a[i]= b[i]+ c[i];
for (i=2; i<N; i++)
  d[i]=a[i-1];;
```

Loop Fusion

- This transformation is useful for
  - Reducing loop overhead
  - Increasing the granularity of work done in a loop
  - Improving locality by combining loops that reference the same array

Loop Peeling

- Remove the first/s or the last/s iteration of the loop into separate code outside the loop
- It is always legal, provided that no additional iterations are introduced.
- When the trip count of the loop is not constant the peeled loop has to be protected with additional runtime tests.

```
for (i=0; i<N; i++)
  A[i] = B[i] + C[i];
```

```
if (N>=1)
  A[0]= B[0] + C[0];
for (i=1; i<N; i++)
  A[i] = B[i] + C[i];
```

Loop Peeling

- This transformation is useful to enforce a particular initial memory alignment on array references prior to loop vectorization.

Loop Unrolling

- Combination of two or more loop iterations together with a corresponding reduction of the trip count.

```
sum =0;
for (i=0; i<N; i++)
  sum += array[i];
```

```
sum =0;
for (i=0; i<N; i+=4) {
  sum += array[i];
  sum += array[i+1];
  sum += array[i+2];
  sum += array[i+3];
}
```

Loop Unrolling

- This transformation is useful
  - To expose more ILP.
  - Reduce overhead instructions
- Register pressure increases, so register spilling is possible
- The unrolled code has a larger size

Unroll and Jam

- Unroll and jam involves partially unrolling one or more loops higher in the nest than the innermost loop, and fusing ("jamming") the resulting loops back together.

```
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];
   }}}
```

```
for (i=0; i<N; i+=2) {
  for (j=0; j<N; j++) {
    for (k=0; k<N; k++) {
      C[i,j] += A[i,k] * B[k,j];
   }}
   for (j=0; j<N; j++) {
    for (k=0; k<N; k++) {
      C[i+1,j] += A[i+1,k] * B[k,j];
   }}
}
```

```
for (i=0; i<N; i+=2) {
  for (j=0; j<N; j++) {
    for (k=0; k<N; k++) {
      C[i,j] += A[i,k] * B[k,j];
      C[i+1,j] += A[i+1,k] * B[k,j];
   }}
}
```

Loop Interchanging

- This transformation switches the positions of one loop that is tightly nested within another loop.

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

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

Loop Interchanging

- This transformation is legal if the outermost loop does not carry any data dependence going from one statement instance executed for i and j to another statement instance executed for i'and j'where i<i' and j>j'

```
for (i=1; i<3; i++){
  for (j=1; j<3; j++){
      A[i,j] = A[i-1, j+1]
```

```
for (i=1; i<3; i++){
  for (j=1; j<3; j++){
      A[i,j] = A[i-1, j-1]
```

Loop Interchanging

```
for (j=1; j<N; j++)
  for (i=2; i<N; i++)
    A[i,j] = A [i-1, j] + B[i]
```

```
for (i=2; i<N; i++)
  for (j=1; j<N; j++)
    A[i,j] = A [i-1, j] + B[i]
```

```
for (i=2; i<N; i++)
   A[i,1:N] = A [i-1, 1:N] + B[1:N]
```

Loop Interchanging

```
for (i=0; i<4; i++){
  a[i] =0;
  for (j=0; j<4; j++)
    a[i]+= b[j][i];
}
```

```
for (i=0; i<4; i++)
  a[i] =0;
for (i=0; i<4; i++)
   for (j=0; j<4; j++)
    a[i]+= b[j][i];
```

```
for (i=0; i<4; i++)
  a[i] =0;
for (j=0; j<4; j++)
   for (i=0; i<4; i++)
    a[i]+= b[j][i];
```

Loop Reversal

- Run a loop backward
- All dependence directions are reversed
- Its is only legal for loops that have no loop carried dependences

```
 for (i=0; i<N; i++){
   a[i]=b[i]+1;
   c[i]=a[i]/2;
}
for (j=0; j<N; j++)
     d[j]+=1/c[j+1];
```

```
 for (i=N-1; i<=0; i--){
   a[i]=b[i]+1;
   c[i]=a[i]/2;
}
for (j=N-1; j<=0; j--)
     d[j]+=1/c[j+1];
```

```
 for (i=N-1; i<=0; i--){
   a[i]=b[i]+1;
   c[i]=a[i]/2;
   d[i]+=1/c[i+1];
}
```

Strip Mining

- Strip mining transforms a singly nested loop into a doubly nested one
- The outer loop steps through the index set in blocks of some size, and the inner loop steps through each block.

```
for (i=0; i<M; i++) {
  A[i] = B[i] +1;
  D[i] = B[i] -1;
}
```

```
for (j=0; j<M; j+=32)
  for (i=j; i< min(j+31, M); i++){
    A[i] = B[i] +1;
    D[i] = B[i] -1;
}
```

Strip Mining

- The block size of the outer block loops is determined by some characteristic of the target machine, such as the vector register length or the cache memory size.

Loop Tiling

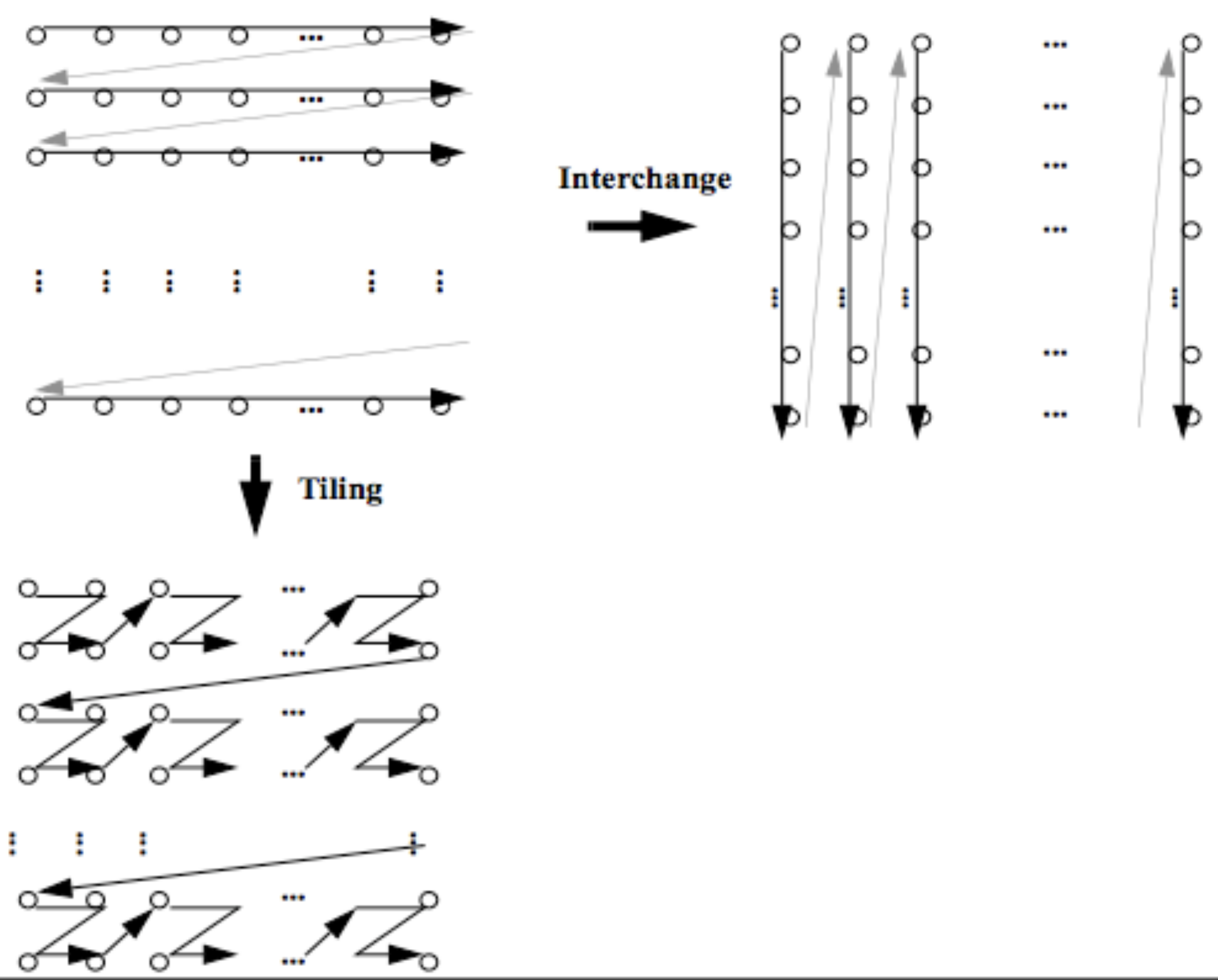
This is a combination of stripming followed by interchange that changes traversal order of a multiply nested loop so that the iteration space is traversed on a tile-by tile basis.

```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        c[i] = a[i,j]*b[i];
```

$\downarrow$

```
for (i=0; i<N; i+=2)
    for (j=0; j<N; j+=2)
        for (ii=i; ii<min(i+2,N); ii++)
           for (jj=j;jj<min(j+2,N); jj++)
                c[ii] = a[ii,jj]*b[ii];
```

Iteration Space and Loop Transformations

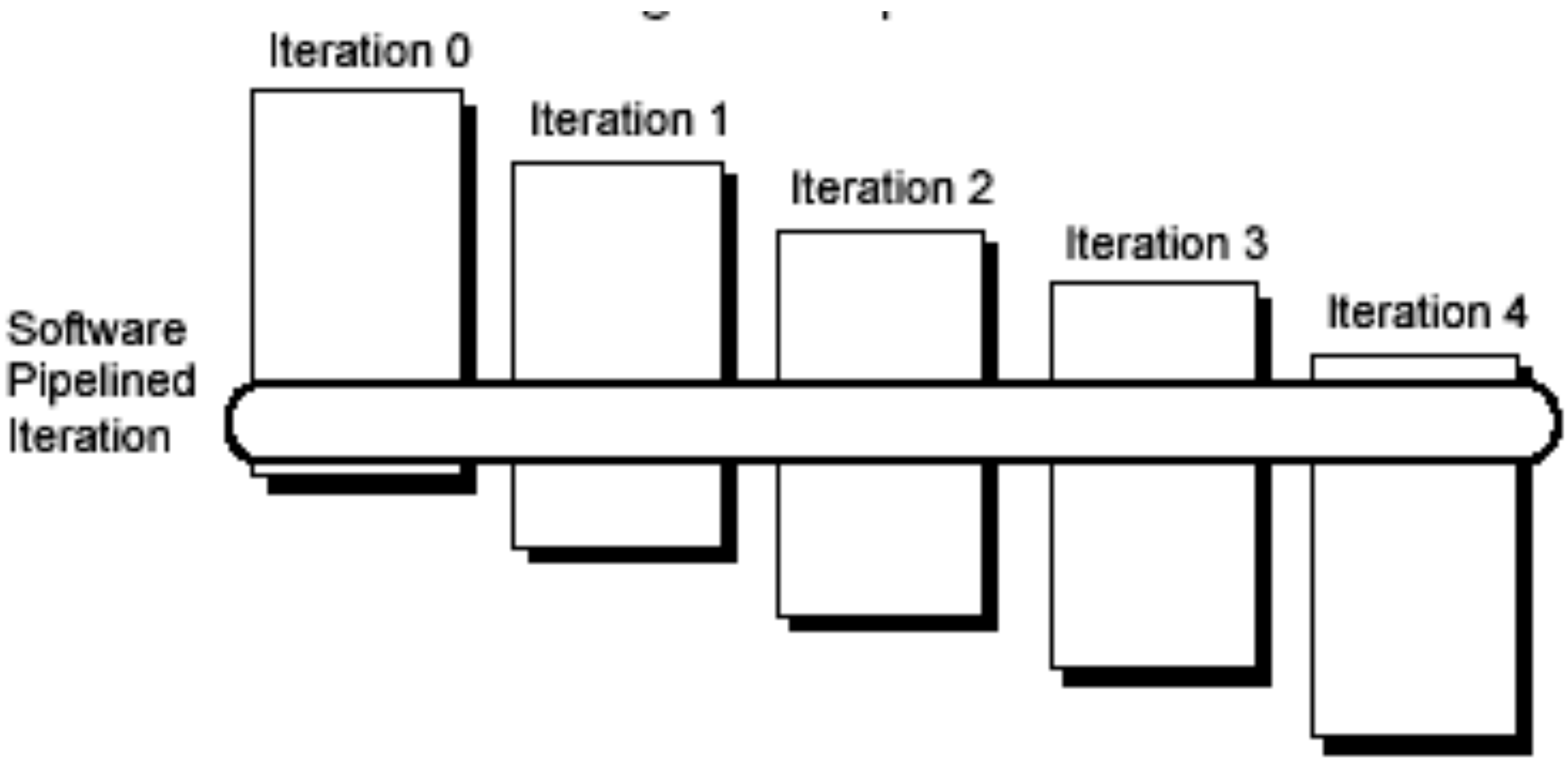


Software Pipelining

- Code reorganization technique to uncover parallelism
- Idea: each iteration contains instructions from several different iterations in the original loop
- The reason: separate the dependent instructions that occur within a single loop iteration
- We need some start-up code (prolog) before the loop begins and some code to finish up after the loop is completed (epilog)

Software Pipelining

- The instructions in a loop are taken from several iterations in the original loop



Software Pipelining

```
Loop:  LD      F0,0(R1)
       ADDD F4,F0,F2
       SD      F4,0(R1)
       DADDUI   R1,R1,#-8
       BNE  R1,R2,Loop
```

```
Loop:  SD F4,16(R1)       ;stores into M[i]
       ADDD F4,F0,F2      ;adds to M[i-1]
       LD F0,0(R1)        ;loads M[i-2]
       DADDUI  R1,R1,#-8
       BNE  R1,R2,Loop
```

```
It i:     LD F0,0(R1)
          ADDD F4,F0,F2
          SD F4,0(R1)
It I+1: LD F0,0(R1)
          ADDD F4,F0,F2
          SD F4,0(R1)
It I+2: LD F0,0(R1)
          ADDD F4,F0,F2
          SD F4,0(R1)
```

Software Pipelining

```
        LD F0,0(R1)
        ADD F4,F0,F2
        LD F0,8(R1)                                  Prolog
        DADDUI  R1,R1,#-16
Loop:   SD F4,16(R1)        ;stores into M[i]
        ADDD F4,F0,F2   ; adds to M[i-1]
        LD F0,0(R1)         ; loads M[i-2]           Loop
        DADDUI  R1,R1,#-8
        BNE  R1,R2,Loop
        SD F4, 16(R1)
        ADD  F4,F0,F2                                Epilog
        SD F4, 16(R1)
```

Software Pipelining

- Notice that the three instructions in the loop are totally independent, as they are working on different elements of the array.

- Because the load and store are separated by two iterations:
  - The loop should run for two fewer iterations
  - The startup code is: LD of iterations 1 and 2, ADDD of iteration 1
  - The cleanup code is: ADDD for last iteration and SD for the last two iterations

Software Pipelining

- Register management can be tricky
- Example shown is not hard: registers that are written in one iteration are read in the next one
- If we have long latencies of the dependences:
  - May need to increase the number of iterations between when we write a register and use it
  - May have to manage the register use
  - May have to combine software pipelining and loop unrolling

Software Pipelining + Loop Unrolling

```
       LD F0,0(R1)
       ADD F4,F0,F2
       LD F0,8(R1)
       DADDUI  R1,R1,#-16
Loop:  SD F4,16(R1)        ;stores into M[i]
       ADDD F4,F0,F2   ; adds to M[i-1]
       LD F0,0(R1)         ; loads M[i-2]
       DADDUI  R1,R1,#-8
       BNE  R1,R2,Loop
       SD F4, 16(R1)
       ADD  F4,F0,F2
        SD F4, 16(R1)
```

```
       LD F6,0(R1)
       ADD F4,F6,F2
       LD F0,8(R1)
       DADDUI  R1,R1,#-16
Loop:  SD F4,16(R1)         ┐
       ADDD F4,F0,F2        │
       LD F0,0(R1)          │
       SD F4,8(R1)          ├  Loop
       ADDD F4,F0,F2        │
       LD F0,-8(R1)         │
       DADDUI  R1,R1,#-16   │
       BNE  R1,R2,Loop      ┘
       SD F4, 16(R1)
       ADD  F4,F0,F2
        SD F4, 16(R1)
```

Software Pipelining vs Loop Unrolling

- Software pipelining consumes less code space
- Both yield a better scheduled inner loop
- Each reduces a different type of overhead:
  - Loop Unroll: branch and counter update code
  - Software Pipelining: reduces the time when the loop is not running at peak speed (only once at the beginning and once at the end)

Software Pipelining with higher latencies

Prolog

```
LD F0,0(R1)
ADD F4,F0,F2
LD F0,8(R1)
ADD F10,F0,F2
LD F0,16(R1)
LD F8,24(R1)
DADDUI  R1,R1,#-32
```

```
Loop:  SD F4,32(R1)        ;stores into M[i]
       ADD F4,F0,F2        ; adds to M[i-2]
       LD F0,0(R1)         ; loads M[i-4]
       SD F10,24(R1)       ;stores into M[i-1]
       ADDD F10,F8,F2      ; adds to M[i-3]
       LD F8,8(R1)         ; loads M[i-5]
       DADDUI  R1,R1,#-16
       BNE  R1,R2,Loop
```

Other Loop Optimizations

- Removal of Loop Invariant Computations
- Induction variable recognition
- Wraparound variable recognition

Loop Invariant Computations

- Calculations that do not change between loop iterations are called loop invariant computations
- These computations can be moved outside the loop to improve performance.

```
for (x=0; x<end; x++)
 array[x] = x * val/3;
```

```
for (x=0; x,100; x++)
   array[x] = x * foo (val);
```

Loop Invariant

```
int FactorialArray[12];

FactorialArray[0] =1;
for (i=1; i<12; i++)
  FactorialArray[i] = FactoriaArray[i-1] * i;


  int FactorialArray[12]= {
       1, 1, 2, 6, 24, 120, 720, 5040,
       40320, 362880, 3628800, 39916800};
```

Induction Variable Recognition

- Induction variables – A variable whose values form an arithmetic progression

```
k=0;
for (i=1; i<N; i++){
  k=k+3
  A[k]= B[k] +1;
}
```

```
for (i=1; i<N; i++){
  A[3*i]= B[3*i] +1;
}
```

Wraparound Variable Recognition

- A variable that looks like an induction variable , but does not quite qualify
- j is a wraparound variable because the values assigned to it are not used until the next iteration of the loop.

```
j=N;
for (i=0; i<N; i++){
  b[i]= (a[j] + a[i])/2;
  j = i;
}
```

```
if (N>=1)
  b[1] = (a[N] + a[1])/2;
for (i=2; i<N; i++){
  b[i]= (a[i-1] + a[i])/2;
}
```