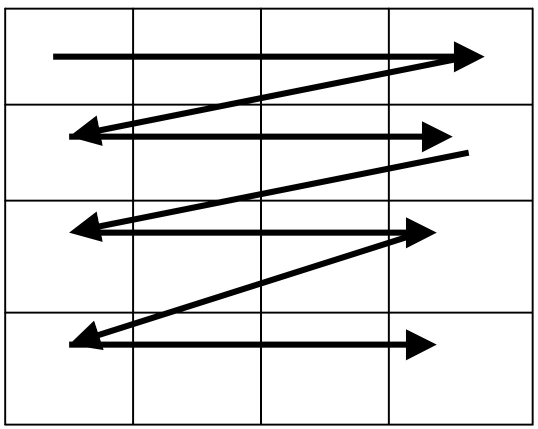
Miss Rate Reduction Techniques

- There are various miss rate reduction techniques such as
  - Larger block size
  - Higher associativity
  - Hardware prefetching
  - Prefetch instructions
- We will discuss <u>compiler optimizations</u> which requires no hardware changes… the magical reduction comes from optimized software…hardware designer's favorite solution!
- The increasing performance gap between processors and main memory has inspired compiler writers to scrutinize the memory hierarchy i.e.to look into compile time optimization options

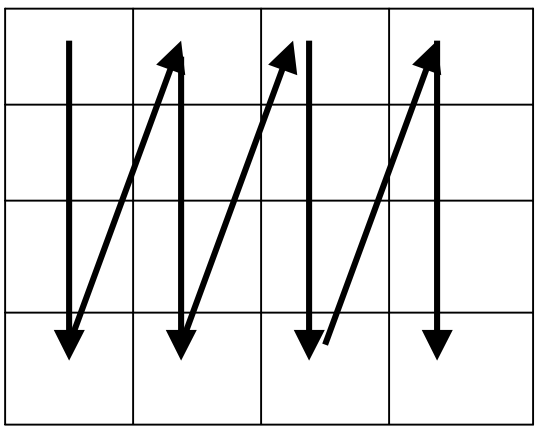
How data stored in memory

- In C data stored row wise in memory; in fortran data stored column wise in memory

C



Fortran



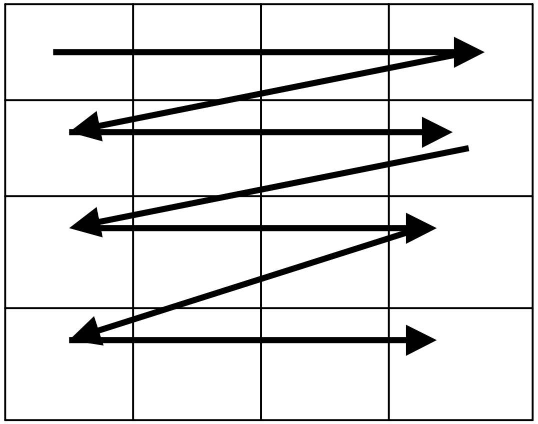
Loop Interchange

- /* before */
  ```
  for (j = 0; j < 100, j = j + 1 )
    for (i = 0; i <5000; i = i +1 )
      X[i][j] = 2* X[i][j] ;
  ```
- /* after* /
  ```
  for (i = 0; i < 5000 ; i = i + 1 )
    for (j = 0; j < 100 ; j = j +1 )
      X[i][j] = 2* X[i][j] ;
  ```
- Original loop accessed data in non sequential order; changing the nesting of the loops can make the code access the data in order it is stored
- This technique reduces misses by improving spatial locality i.e. maximized use of data in cache block before it is discarded
- Need to know how data stored in C or fortran

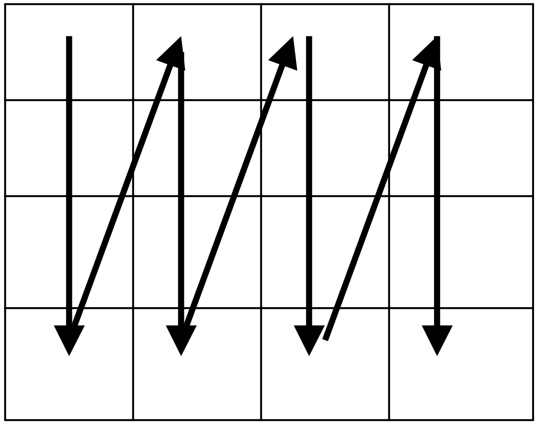
Loop Interchange (cont..)

- The original code would skip through memory in stride of 100 words; you always want stride of 1
- In C data stored row wise in memory; in fortran data stored column wise in memory

C



fortran



Loop Fusion

- Some programs have separate sections of code that accesses the same arrays with the same loops and perform different computations on the common data
- By "fusing" the code into single loop, the data that are fetched into the cache can be used repeatedly before being swapped out
- This approach utilized <u>temporal locality</u> principle of cache design

Loop Fusion (cont….)

```
/*before*/
    for (i=0; i<N; i = i+1)
    for (j=0; j<N; j = j+1)
            A[i][j] =
    1/B[i][j]*C[i][j] ;

     for (i=0; i<N; i = i+1)
    for (j=0; j<N; j = j+1)
            D[i][j] = A[i][j] +
    C[i][j] ;
```

```
/*after*/
    for (i=0; i<N; i = i+1)
    for (j=0; j<N; j = j+1)
    {
            A[i][j] =
    1/B[i][j]*C[i][j] ;


            D[i][j] = A[i][j] +
    C[i][j] ;
     }
```

The original code will take all the misses to access array A and C twice, once in the 1st loop and then again in the 2nd loop. In the fused loop the 2nd statement freeloads on the cache accesses of the first statement.

Blocking

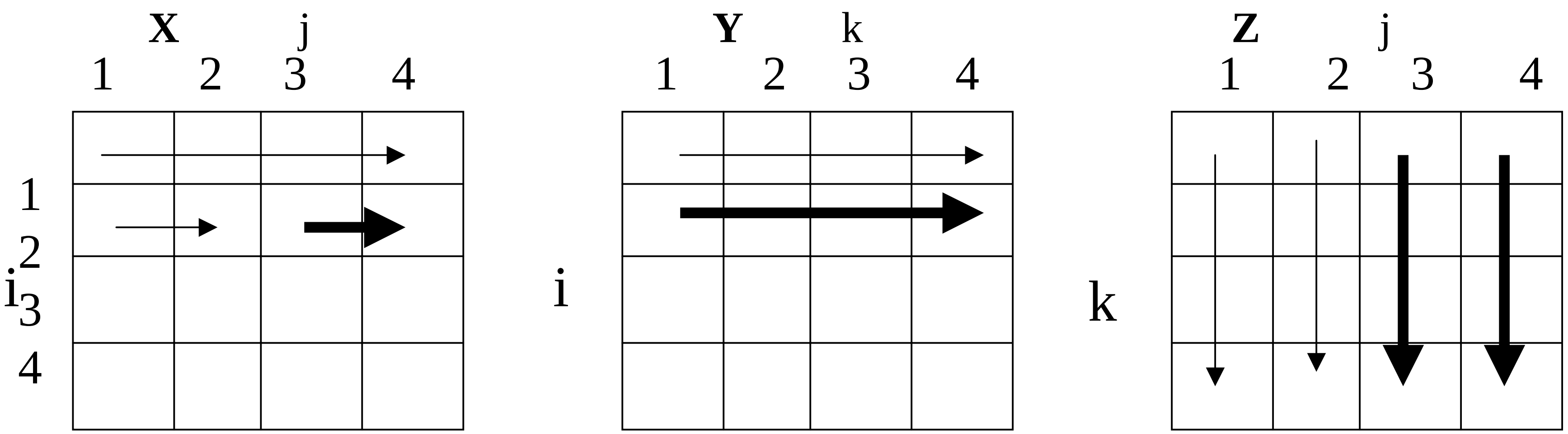
- The most famous of cache optimization
- Tried to reduce misses by improving mostly temporal locality
- Dealing with multiple arrays where some arrays are accessed by rows and some by columns
- Since both rows and columns are used in every iteration of the loop storing accessing by rows or columns doesn’t help (loop interchange doesn’t help)
- Instead of operating on entire rows or columns of an array, blocked algorithms operate on submatrices or blocks.
- Goal is to maximize accesses to the data loaded into the cache before it is replaced

Blocking (cont…)

```
! before
    do i = 1,n
    do j = 1, n
    do k = 1,n
          X(i,j) = X(i,j) + Y(i,k)*Z(k,j)
    end do
    end do
    end do
```

- The two inner loop read all N by N elements of Z
- Access the same N elements in a row of Y repeatedly
- Write one row of N elements of X

Blocking (cont…)



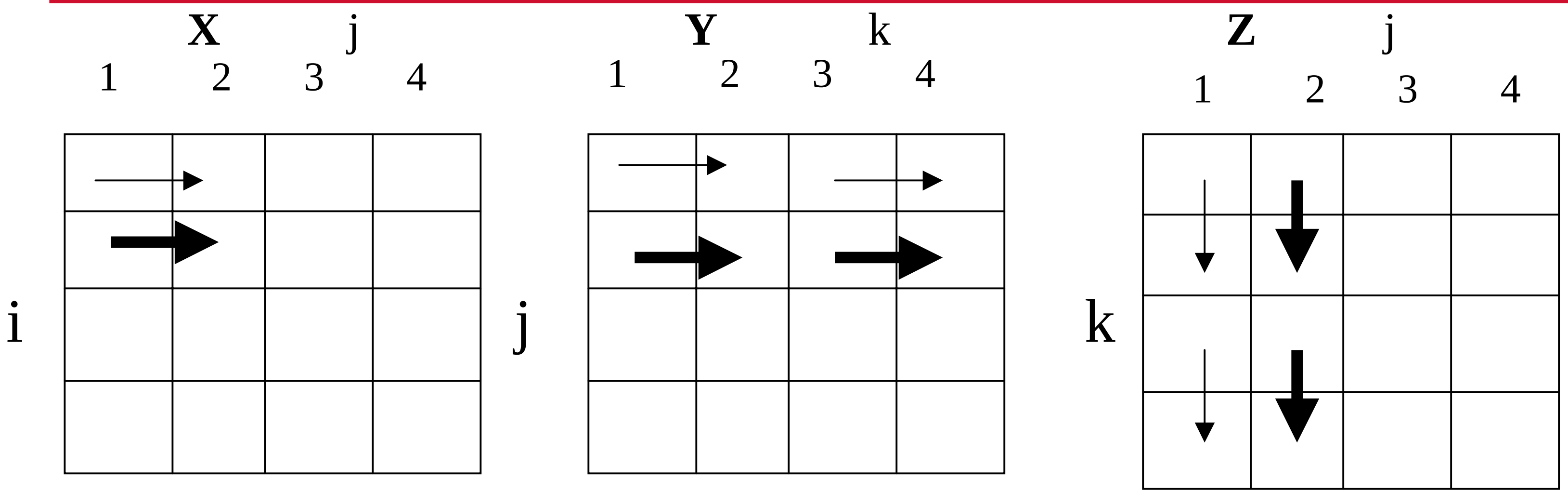
- White empty boxes are not yet touched
- Thin arrow means older access
- Bold arrow means newer access
- If all the data fit in cache (three N X N matrices), there is no problem
- Otherwise in worst case there would be (2N + 1) elements read from memory to calculate one element of X
- For N^2 element of X, in worst case, there would be $(2N^3 + N^2)$ read from memory

Blocking (cont…)

- To ensure that the elements accessed fit in the cache, the original code is changed to compute on a **submatrix** of size **nB** by **nB**
- This is done by having the loops compute in step size of nB rather than going from beginning to end of X and Z

```
!after
    do ib = 1, n, nB
    do jb = 1, n, nB
    do kb = 1,n, nB
        do i = ib, min(n,ib+nB-1)
        do j = jb, min(n,jb+nB-1)
        do k = kb, min(n,kb+nB-1)
                X(i,j) = X(i,j) + Y(i,k)*Z(k,j)
        end do
        end do
        end do
    end do
    end do
    end do
```

Blocking (cont…)



- White empty boxes are not yet touched
- Thin arrow means older access
- Bold arrow means newer access
- In contrast to before smaller number of elements are accessed
- This exploits combination of spatial and temporal locality depending on fortran (Y temporal and Z spatial) or C (Y spatial and Z temporal)
- nB is the blocking factor

Blocking (cont…)

loop equation : <u>X(i,j) = X(i,j) + Y(i,k)*Z(k,j)</u>

first pass over i, j, k :

ib = 1,4,2
jb = 1,4,2
kb = 1,4,2

i = 1,2
j=1,2
k=1,2

X11 = Y11*Z11+Y12*Z21
X12 = Y11*Z12+Y12*Z22
X21 = Y21*Z11+Y22*Z21
X22 = Y21*Z12+Y22*Z22

Blocking (cont…)

loop equation : $\underline{X(i,j) = X(i,j) + Y(i,k)*Z(k,j)}$

second pass over i, j, k :

ib = 1,4,2

jb = 1,4,2

kb = 3, 4

i = 1,2

j=1,2

k=3,4

$X11 = X11 + Y13*Z31+Y14*Z41$

$X12 = X12 + Y13*Z32+Y14*Z42$

$X21 = X21 + Y23*Z31+Y24*Z41$

$X22 = X22 + Y23*Z32+Y24*Z42$

Blocking (cont…)

- Continue blocking algorithm….