

# Locality / Tiling

---

María Jesús Garzarán  
CS 498: Compiler Optimizations  
Fall 2006

University of Illinois at Urbana-Champaign



TM

# Roadmap

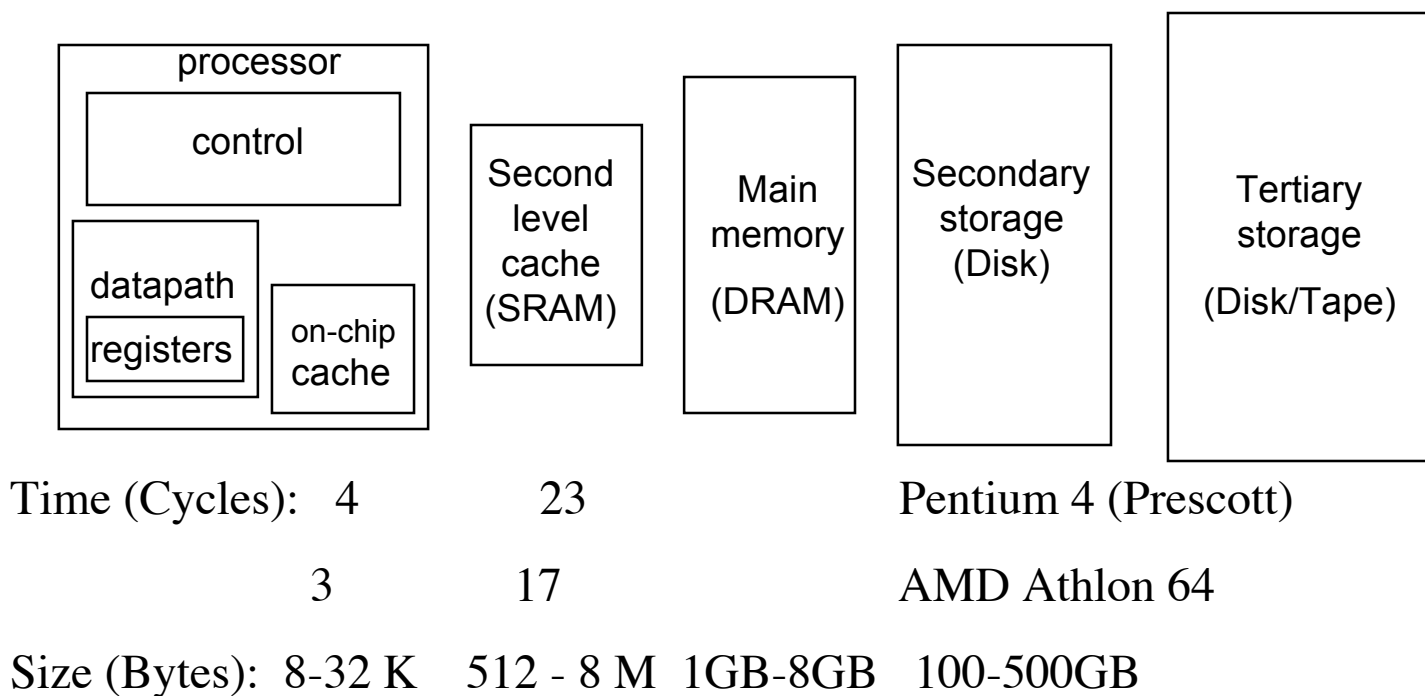
---

- ◆ We focus on program locality on loops
  1. Memory Hierarchy
  2. Classification of Cache misses
  3. Loop Interchange
  4. Loop Tiling
  5. Tile Size Selection based on the article:  
“Tile Selection using cache organization and data layout”, by S. Coleman and K. McKinley, PLDI 1995, pages 279 - 280  
[www.cs.utexas.edu/users/mckinley/papers/pldi-1995.pdf](http://www.cs.utexas.edu/users/mckinley/papers/pldi-1995.pdf)



# Memory Hierarchy

- ◆ Most programs have a high degree of **locality** in their accesses
  - Spatial locality: accessing things nearby previous accesses
  - Temporal locality: accessing an item that was previously accessed
- ◆ Memory Hierarchy tries to exploit locality



# Background: Cache Misses

---

- Compulsory Misses: First reference to a data
- Capacity Misses: The amount of data accessed is larger than the cache. They are computed with respect to a fully associative LRU cache.
- Conflict or Interference Misses: A cache line that contains data that will be used is replaced by another cache line.
  - Self-interference misses: an element of the same array causes the interference miss.
  - Cross-interference misses: an element of a different array causes the interference miss.
- Coherence Misses: Appear in parallel programs due to sharing



# Example 1

---

- ◆ Cache Line Size (CLS) = 32 bytes
- ◆ Cache Size (CS) = 64 lines
- ◆ Direct mapping
- ◆ Each element of a is 4 bytes long

```
real a (1024, 100)
...
do i=1, 1024
  do j=1, 100
    a(i,j) = a(i,j) + 1
  end do
end do
```

$a_{11}$	$a_{12}$	$a_{13}$	$a_{14}$	$a_{15}$ ...
$a_{21}$	$a_{22}$	$a_{23}$	$a_{24}$	$a_{25}$ ...
$a_{31}$	$a_{32}$	$a_{33}$	$a_{34}$	$a_{35}$ ...
	...			
$a_{81}$	$a_{82}$	$a_{83}$	$a_{84}$	$a_{85}$ ...
$a_{91}$	$a_{92}$	$a_{93}$	$a_{94}$	$a_{95}$ ...

- ◆ Cache misses =  $100 * 1024 = 102,400$



# Example 1. Loop interchange

---

```
real a (1024, 100)
...
do j=1, 1024
  do i=1, 100
    a(i,j) = a(i,j) + 1
  end do
end do
```

$a_{11}$	$a_{12}$	$a_{13}$	$a_{14}$	$a_{15}$ ...
$a_{21}$	$a_{22}$	$a_{23}$	$a_{24}$	$a_{25}$ ...
$a_{31}$	$a_{32}$	$a_{33}$	$a_{34}$	$a_{35}$ ...
	...			
$a_{81}$	$a_{82}$	$a_{83}$	$a_{84}$	$a_{85}$ ...
$a_{91}$	$a_{92}$	$a_{93}$	$a_{94}$	$a_{95}$ ...

- ◆ Takes advantage of spatial locality to reduce #misses
- ◆ Cache misses = 102,400 / 8, number of cache lines occupied by the array a
- ◆ These are compulsory misses and, unless prefetch is used, it is impossible to eliminate them



# Example 2

---

```
do j=1, 100
  do i=1, 4096
    a(i) = a(i) * a(i)
  end do
end do
```

- ◆ Cache misses =  $(4096 / 8) * 100$
- ◆ First access to  $a(1)$ ,  $a(9)$ ,  $a(17)$ ,  $a(25)$ , ... ,  $a(4089)$  will cause a compulsory miss
- ◆ After accessing elements  $a(1..512)$  the cache is full, and a cache line has to be written back to memory before a new line is brought to cache
- ◆ Each element  $a(i)$  is reused 100 times --> We can apply tiling to reduce the number of cache misses



## Example 2: Tiling

---

```
do ii=1, 4096, B
  do j=1, 100
    do i=ii, min(ii+B-1,4096)
      a(i) = a(i) + a(i)
    end do
  end do
end do
```



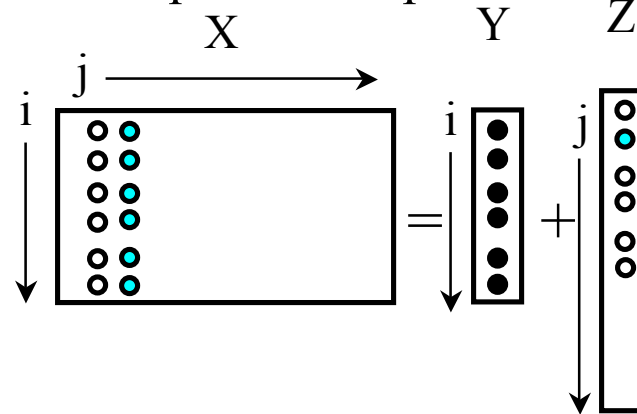
- ◆ The cache can accommodate 512 elements of  $a$ .
- ◆ Thus, if we choose  $B=512$ , cache misses =  $4096 / 8$ , which are the compulsory misses



# Example 3

- Reorder the loops below to improve temporal and spatial locality

```
do j=1 to M
do i=1 to N
x(i,j) = y(i) + z(j)
```



$$\text{Compulsory misses} = N \cdot M / \text{CLS} + N / \text{CLS} + M / \text{CLS}$$

CLS = Cache Line Size; CS = Cache Size in Lines

Fully Associative Cache

N and M are multiples of the CLS

If  $(CS < 1 + N / \text{CLS} + 1)$

$$\text{Cache misses} = N \cdot M / \text{CLS} + N / \text{CLS} \cdot M + M / \text{CLS}$$



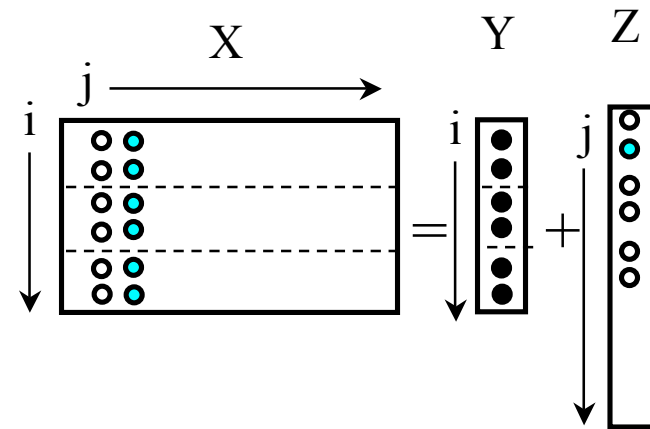
# Example 3

- Reorder the loops below to improve temporal and spatial locality

```
do j=1 to M
  do i=1 to N
    x(i,j) = y(i) + z(j)
```

- Tiling

```
do ii=1 to N, B
  do j=1 to M
    do i=ii to min(N,ii+B-1)
      x(i,j) = y(i) + z(j)
```



N and M are multiples of CLS

N is a multiple of B

B is a multiple of CLS

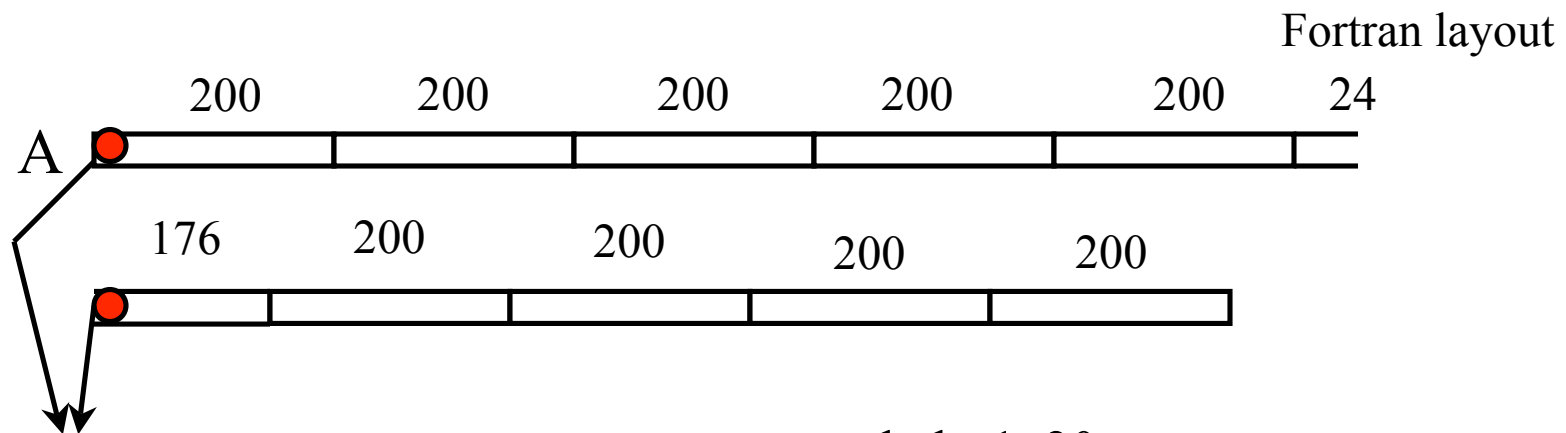
If  $(CS < 1 + B/CLS + M/CLS)$

$$\text{Cache misses} = N * M / CLS + N / CLS + M / CLS * N / B$$



# Interference or Conflict misses

- ◆ They are due to the cache replacement policy that maps data to the same location.
- ◆ **Self-interference misses. Example:**  
array A(200,10)  
cache size 1024 elements of A, direct mapped



A(1) and A(1025) occupy  
the same cache line

```
do k=1, 20  
  do j=1, 10  
    do i=1,25  
      ... A(i,j) ...  
    
```



# Tile Selection

---

“Tile Selection using cache organization and data layout”,  
by S. Coleman and K. McKinley, PLDI 1995, pages 279 - 280

[www.cs.utexas.edu/users/mckinley/papers/pldi-1995.ps.gz](http://www.cs.utexas.edu/users/mckinley/papers/pldi-1995.ps.gz)

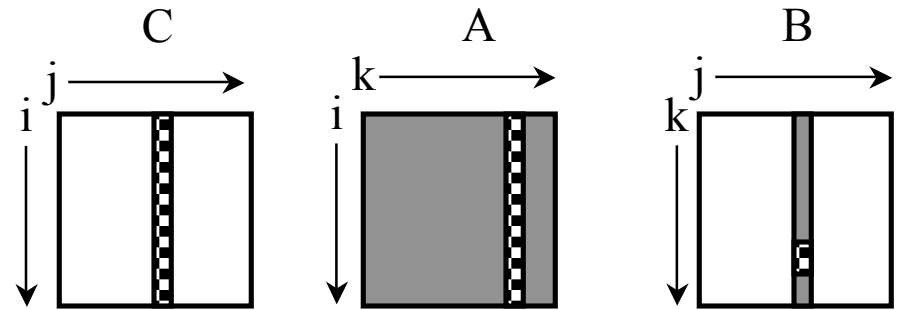


# Matrix-matrix Multiplication

## matrix multiplication

```

do j=1,N
  do k=1,N
    r = B(k,j)
    do i=1,N
      C(i,j) = A(i,k) * r
    
```

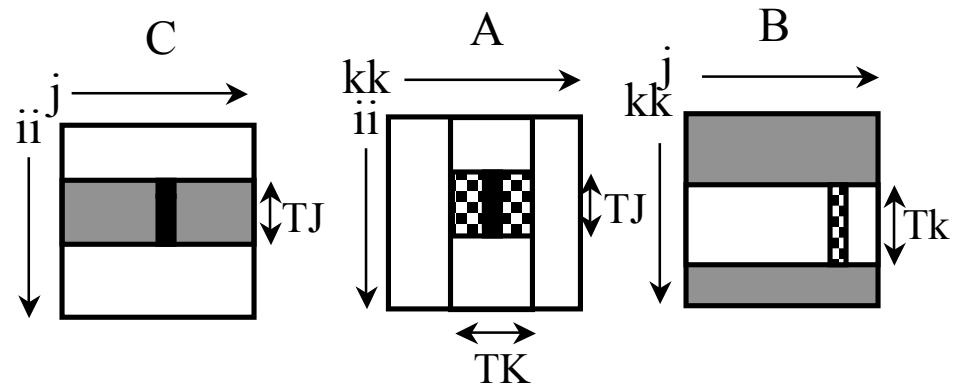


- $2*N + N^2$  elements accessed per iteration of the  $j$  loop
- Between each reuse of an element of  $A$ 
  - $N$  elements of  $C$  accessed in  $i$  loop
  - $N$  elements of  $B$  accessed in  $k$  loop
  - $N^2 - 1$  of array  $A$

## tiled matrix multiplication

```

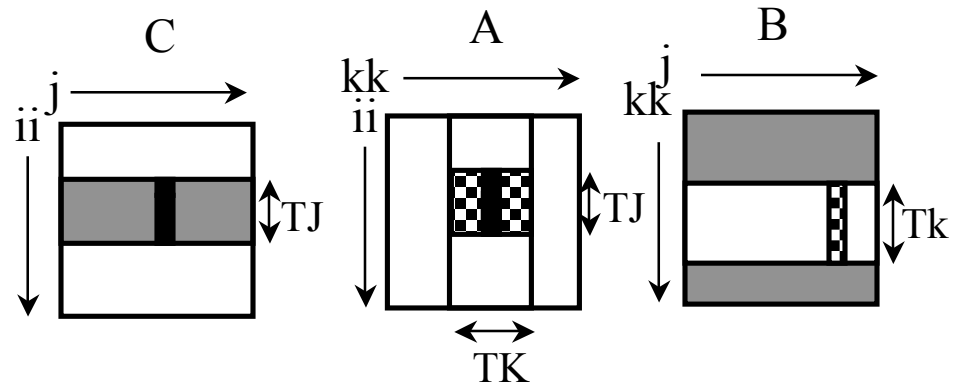
do kk=1, N, TK
  do ii =1, N, TJ
    do j=1,N
      do k=kk,min(kk+TK-1, N)
        r = B(k,j)
        do i=ii,min(ii+TJ-1,N)
          C(i,j) = A(i,k) * r
        
```



## tiling matrix multiplication

```

do kk=1, N, TK
  do ii =1, N, TJ
    do j=1,N
      do k=kk,min(kk+TK-1, N)
        r = B(k,j)
        do i=ii,min(ii+TJ-1,N)
          C(i,j) = A(i,k) * r
        
```



	Reuse Factor			Footprint		
	I	K	J	I	K	J
<b>B(k,j)</b>	TJ	0	0	1	1	TK
<b>A(i,k)</b>	0	0	N	1	TJ	TK*TJ
<b>C(i,j)</b>	0	TK	0	1	TJ	TJ

The largest tile with the most reuse in  $j$  is  $A(i,k)$ .

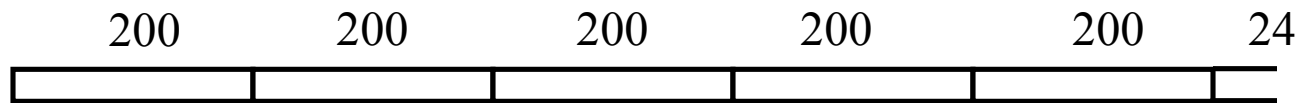
The target is to choose  $TK$  and  $TJ$  so that the  $TK*TK$  submatrix of  $Y$  will be in cache, and there is enough room in cache for the working set of the  $j$  loop ( $TK*TJ+TK_{\frac{1}{4}}TJ$ )



# Tile Size Selection. Self Interference misses

---

- ◆ Array of 200x200 elements
- ◆ Cache of 1024 elements; CS = Cache Size; CLS = Cache Line Size, Direct mapped
- ◆ Number of complete columns that fit in the cache is simply:
  - $\text{ColsPerSet} = \lfloor \text{CS}/N \rfloor$ ; N column dimension -- consecutive stored dimension



- Essegir[1] selects this tile size (200 x 5)

- ◆ [1] K. Essegir. Improving data locality for caches. Mater's thesis. Dept. of Computer Science, Rice University, September 1993



# Tile Selection. Self Interference Misses

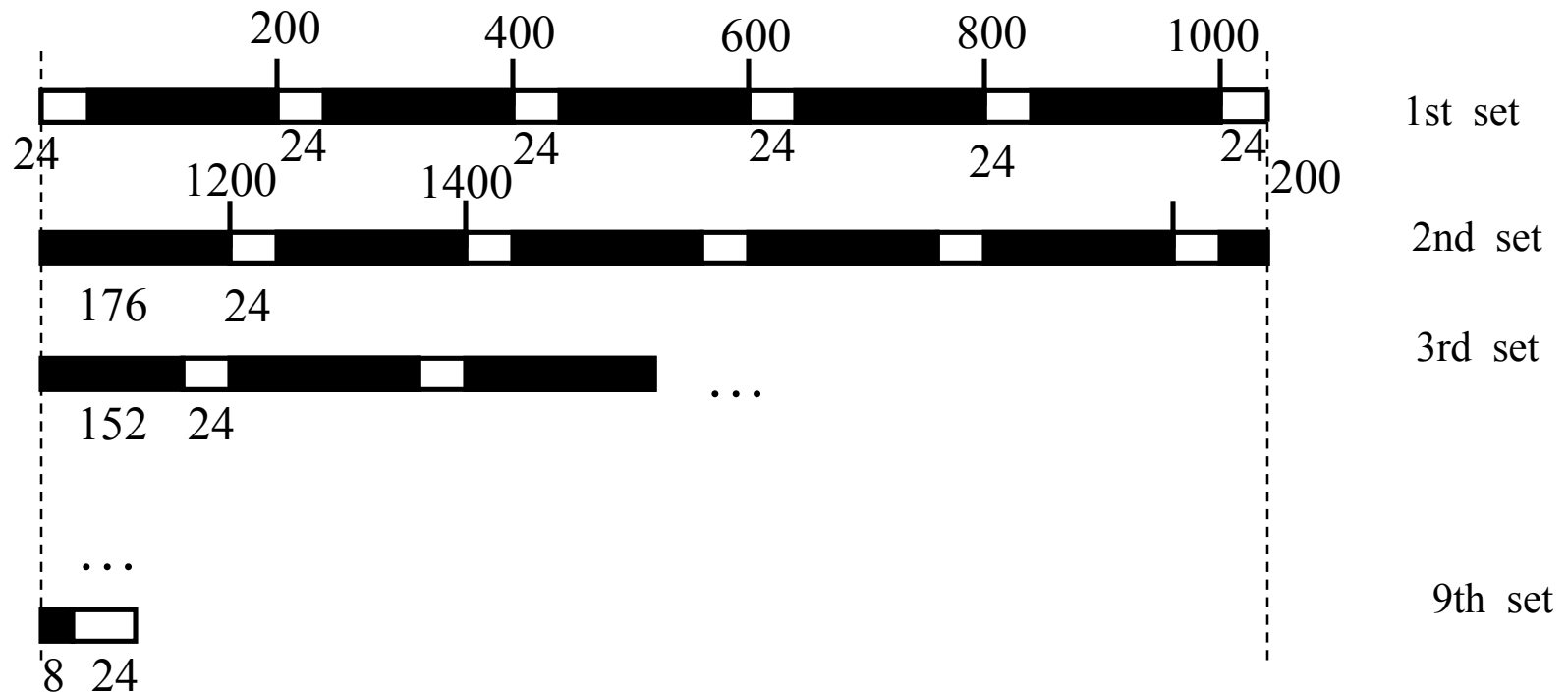
---

- ◆ Use of the Euclidean algorithm to generate potential column dimensions of the tile
  - The Euclidean algorithms find the g.c.d.(a,b) ( $a > b$ ). It computes:  
 $a = q_1 * b + r_1$   
 $b = q_2 * r_1 + r_2$   
....  
 $c = q_{k+1} * r_k + r_{k+1}$
- ◆ For our purposes  $a = CS$  (Cache Size) and  $b = N$  (column dimension --the consecutive stored dimension).  
 $1024 = 5 * 200 + 24$   
 $200 = 8 * 24 + 8$   
Since 8 divides 24,  $8 = \text{gcd}(1024, 200)$ , and the algorithm finishes.
- ◆ Each remainder is a potential column size.
- ◆ For each potential column size, we need to find the size of the row (the other dimension of the tile).



# Tile Size Selection. Self Interference misses

Column Size = 24 Row Size? 41



Starting position of 1st and 2nd set differ by  $\text{SetDiff} = N - r1 = 200 - 24 = 176$

The difference between subsequent sets will eventually become  $\text{Gap} = N \bmod r1 = 200 \bmod 24 = 8$ . The row size is determined by the point at which the difference changes to Gap. (Algorithm in Figure 4 of the paper).



# Tile Selection. Cache Line Size

---

- ◆ Choose column sizes that are multiples of the cache lines in terms of elements, CLS

- ◆ 
$$\text{colSize} = \begin{cases} \text{Colsize} & \text{if } (\text{Colsize} \bmod \text{CLS} = 0), \text{ or} \\ & \text{if } (\text{Colsize} = \text{column length}) \\ \lfloor \text{Colsize}/\text{CLS} \rfloor & \text{otherwise} \end{cases}$$



# Tile Selection. Cross-Interferences

---

- ◆ Use of footprints to determine amount of data accessed
- ◆ On an iteration of a  $j$  loop, we access  $TK \times TJ$  elements of  $A$ ,  $TJ$  of  $C$ , and  $TK$  of  $B$ .
- ◆ If we completely fill up the cache with  $TK \times TJ$  elements of  $A$ , there will be cross interference misses (CIM)  $CIM = 2 \times TJ + TK$ 
  - $2 \times TJ$  (Array  $A$  interferes with  $C$  and  $C$  interferes with  $A$ )
  - $TK$  (Array  $A$  interferes with  $B$ )
- ◆ Cross interference rate (CIR)
  - $CIR = CIM / (TJ \times TK)$  (2)
- ◆ Selection of Tiles so that the Working Sets (WS) fit in the cache:  
 $TJ \times TK + TJ + 1 \times CLS < CS$  (3)



- ◆ Use of the CIR and working set to distinguish between the tile sizes generated based on the algorithm on self-interference

# Tile Selection Algorithm

---

- ◆ The algorithm terminates if:
    - the column length evenly divides the cache (powers of 2 arrays)
    - tiles are larger than the arrays (there is no need to tile)
  - ◆ Initially
    - InitialTile = BestTile is such that BestCol = N and BestRow = CS/N
  - ◆ While loop that iterates finding potential rows without self-interference for each of the Euclidean numbers (ColumnSize = Euclidean number)
    - Find the RowSize that has no self-interference for ColumnSize
    - Adjust ColumnSize to be a multiple of CLS
    - Tile=Rowsize,ColumnSize
    - If ((WS (Tile) > WS (BestTile)) and (3)  
(WS (Tile) < CS ) and  
(CIR(Tile) < CIR(BestTile))) (2)  
then  
BestTile = Tile
  - ◆ If (WS(BestTile) > CS) then
    - reduce ColumnSize of the InitialTile by CLS
- until WS(Tile) <= CS



- Algorithm in Figure 5 of the paper.

# Set Associativity & TLB

---

- ◆ Associativity does not affect the tile selected for a particular cache size
- ◆ The tile size needs to be constrained so that the number of rows is smaller than the number of page table entries in the TLB

