清華大學
Tsinghua University

Writing Cache Friendly Code

——Slides derived from those by Randy Bryant …

问题1

- **Example: cold cache, 4-byte words, 4-word cache blocks**

```
int sum_array_rows(int a[M][N])
{
  int i, j, sum = 0;

  for (i = 0; i < M; i++)
     for (j = 0; j < N; j++)
        sum += a[i][j];
  return sum;
}
```

```
int sum_array_cols(int a[M][N])
{
  int i, j, sum = 0;

  for (j = 0; j < N; j++)
     for (i = 0; i < M; i++)
        sum += a[i][j];
  return sum;
}
```

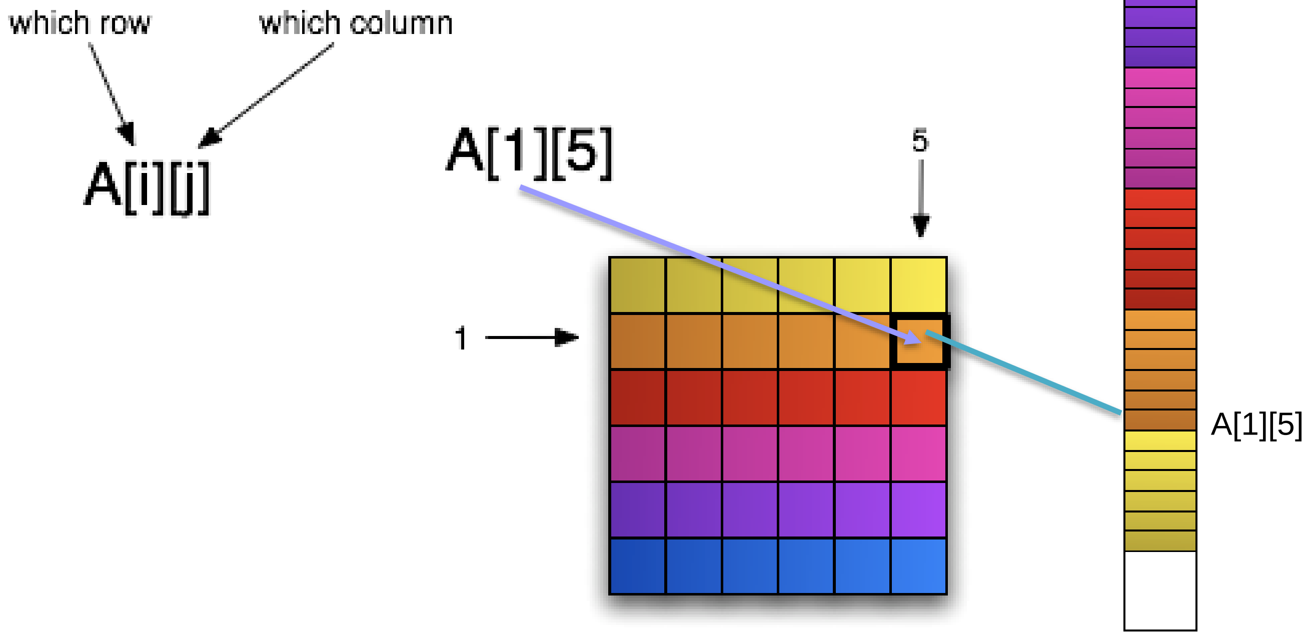
Miss rate =?

Miss rate =?

Layout of C Arrays in Memory

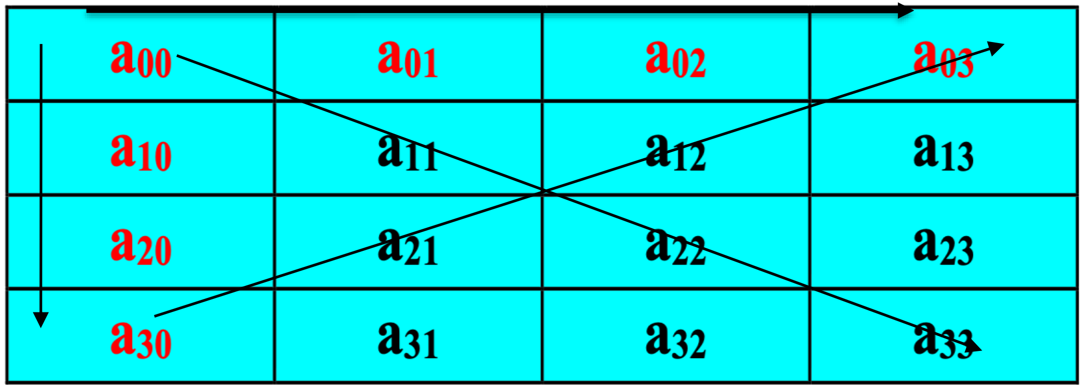
C arrays allocated in **row-major** order

- each row in contiguous memory locations



问题2

- 一个*n*×*n*的二维数组，按行、列、对角线和反对角线访问，并且在不同的变址位移量情况下，都能实现无冲突访问?



a_{00}	a_{01}	a_{02}	a_{03}
a_{10}	a_{11}	a_{12}	a_{13}
a_{20}	a_{21}	a_{22}	a_{23}
a_{30}	a_{31}	a_{32}	a_{33}

Cache 性能

$$CPU\ time = \frac{IC \times \left(CPI_{Exec} + \frac{MemAccess}{Inst} \times MissRate \times MissPenalty\right)}{Clock\ rate}$$

$$CPU\ time = \frac{IC \times \left(\frac{ALUops}{Inst} \times CPI_{ALUops} + \frac{MemAccess}{Inst} \times AMAT\right)}{Clock\ rate}$$

$$\boxed{\begin{aligned}&AMAT = Hit\ time + Miss\ Rate \times Miss\ Penalty\\&= \%\ instructions \times (Hit\ time_{Inst} + Miss\ Rate_{inst} \times Miss\ Penalty_{Inst})\\&+ \%\ data \times (Hit\ time_{Data} + Miss\ Rate_{Data} \times Miss\ Penalty_{Data})\end{aligned}}$$

Cache性能

- Cache 优化
  - 1. 减少缺失率（miss rate）
  - 2. 减少缺失代价（miss penalty）
  - 3. 较少判断命中时间（hittime）

$$AMAT = HitTime + MissRate \times MissPenalty$$

Cache Organization?

- Change Block Size
- Change Cache Size
- Change Cache Internal Organization
- Change Associativity
- Change Compiler
- Which of 3Cs is obviously affected?

Cache的设计空间

- Larger cache size
  - + reduces capacity and conflict misses
  - - hit time will increase
- Higher associativity
  - + reduces conflict misses
  - - may increase hit time
- Larger block size
  - + reduces compulsory and capacity misses
  - + exploit burst transfers in memory and on buses
  - - increases conflict misses and miss penalty

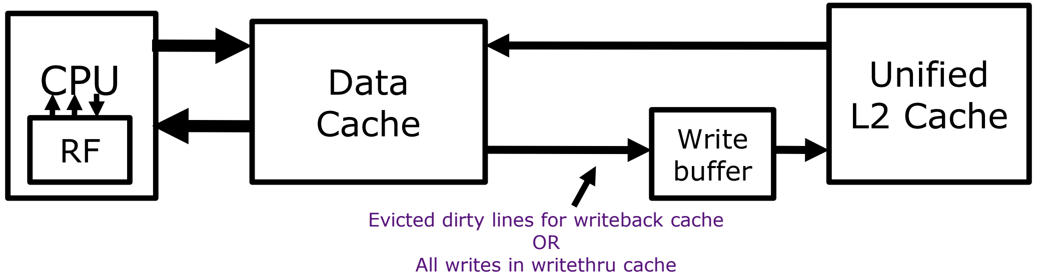
减少缺失率(MissRate)

$$CPU\ time = \frac{IC \times \left(CPI_{Exec} + \frac{MemAccess}{Inst} \times \boxed{MissRate} \times MissPenalty \right)}{Clock\ rate}$$

1. Larger Cache => Reduce Capacity
2. Larger Block Size => Reduce Compulsory
3. Higher Associativity => Reduce Confilcts
4. Way Prediction & Pseudo-Associativity
5. Compiler Optimizations

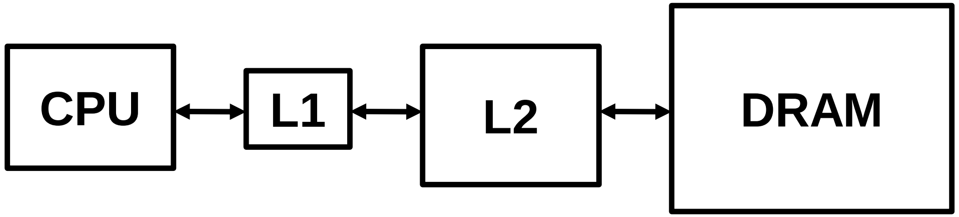
减少缺失代价(MissPenalty)

- 1. Multilevel Caches
- 2. Critical Word First and Early Restart
- 3. Giving Priority to Read Misses over Writes
- 4. Merging Write Buffer
- 5. Victim Caches
- * Reducing Read Miss Penalty—Write Buffer


CPU
RF
Data Cache
Write buffer
Unified L2 Cache
Evicted dirty lines for writeback cache
OR
All writes in writethru cache

$$CPU\ time = \frac{IC \times \left(CPI_{Exec} + \frac{MemAccess}{Inst} \times MissRate \times \boxed{MissPenalty} \right)}{Clock\ rate}$$

多级Cache



- Inclusive multilevel cache:
  - Inner cache holds copies of data in outer cache
  - External access need only check outer cache
  - Most common case
- Exclusive multilevel caches:
  - Inner cache may hold data not in outer cache
  - Swap lines between inner/outer caches on miss
  - Used in AMD Athlon with 64KB primary and 256KB secondary cache

减少命中时间（HitTime）

- Small and Simple Caches
- Avoiding Address Translation
- Reducing Write Hit Time—Pipelining Cache Write

$$AMAT = \boxed{HitTime} + MissRate \cdot MissPenalty$$

Technique	Miss-Rate	Miss-Penelty	HitTime	Complexity
Larger Block Size	+	-		0
Higher Associativity	+		-	1
Victim Caches	+			2
Pseudo-Associative Caches	+			2
HW Prefetching of Instr/Data	+	+		2
Compiler Controlled Prefetching	+	+		3
Compiler Reduce Misses				0
Priority to Read Misses		+		1
Early Restart & Critical Word 1st		+		2
Non-Blocking Caches		+		3
Second Level Caches		+		2
Better memory system		+		3
Small & Simple Caches	-		+	0
Avoiding Address Translation			+	2
Pipelining Caches			+	2

Instruction vs. Data Caches

- Separate or Unified?

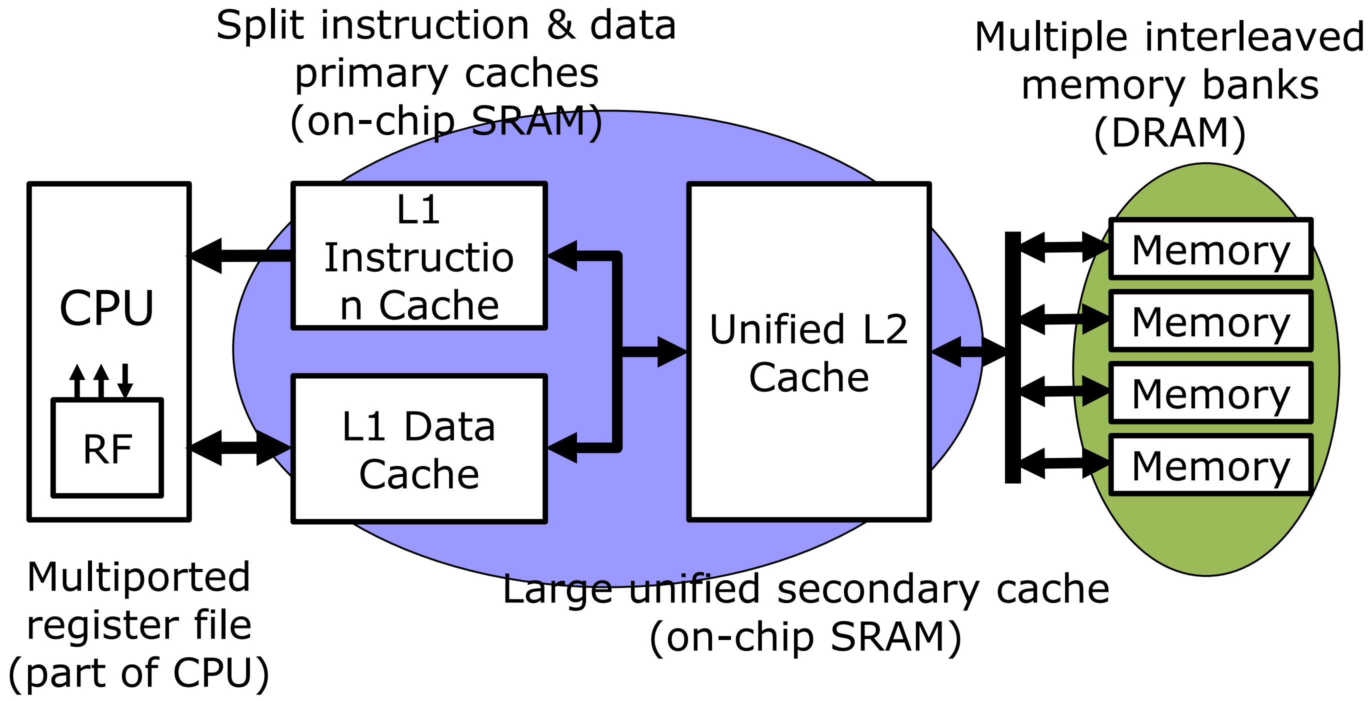
- Unified:
  + Dynamic sharing of cache space: no over provisioning that might happen with static partitioning (i.e., split I and D caches)
  -- Instructions and data can thrash each other (i.e., no guaranteed space for either)
  -- I and D are accessed in different places in the pipeline. Where do we place the unified cache for fast access?

- First level caches are almost always split
  - Mainly for the last reason above
- Second and higher levels are almost always unified

Multi-level Caching in a Pipelined Design

- First-level caches (instruction and data)
  - Decisions very much affected by cycle time
  - Small, lower associativity
  - Tag store and data store accessed in parallel
- Second-level caches
  - Decisions need to balance hit rate and access latency
  - Usually large and highly associative; latency not as important
  - Tag store and data store accessed serially
- Serial vs. Parallel access of levels
  - Serial: Second level cache accessed only if first-level misses
  - Second level does not see the same accesses as the first
    - First level acts as a filter (filters some temporal and spatial locality)
    - Management policies are therefore different

存储器系统层次结构


Split instruction & data primary caches (on-chip SRAM)
Multiple interleaved memory banks (DRAM)
CPU
RF
L1 Instruction Cache
L1 Data Cache
Unified L2 Cache
Memory
Memory
Memory
Memory
Multiported register file (part of CPU)
Large unified secondary cache (on-chip SRAM)

Performance of 1-Level Split Cache

$$CPI^{stall} = \frac{\text{instruction stall cycles}}{\text{stalls}} \times \frac{\text{instruction stalls}}{\text{instruction access}} \times \frac{\text{instruction accesses}}{\text{instruction}}$$

$$+ \frac{\text{data stall cycles}}{\text{stalls}} \times \frac{\text{data stalls}}{\text{data access}} \times \frac{\text{data accesses}}{\text{instruction}}$$

$$= \text{instruction miss penalty} \times \text{instruction miss rate} \times \frac{\text{1 instruction access}}{\text{instruction}}$$

$$+ \text{data miss penalty} \times \text{data miss rate} \times \frac{IC^{load} + IC^{store}}{IC}$$

Typical values

Instruction Miss Penalty = Data Miss Penalty = 50 cycles per stall

Instruction Miss Rate = 0.5%

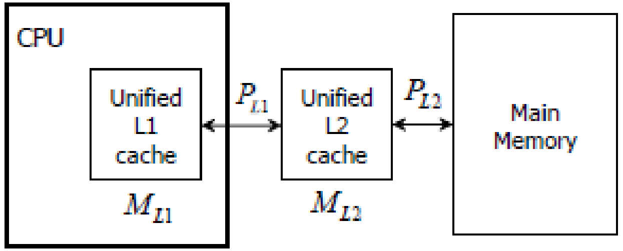
Data Miss Rate = 5.0%

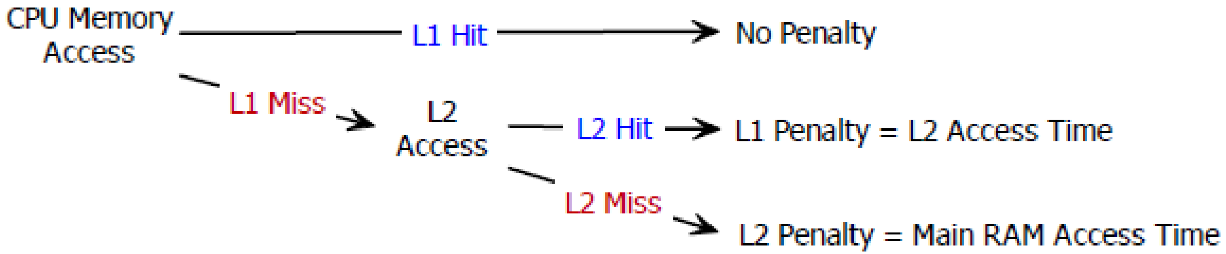
IC^{load} = 25%

IC^{store} = 15%

$$\boxed{\begin{array}{l} CPI^{stall} = 50 \times 0.005 \times 1 + 50 \times 0.05 \times 0.40 = 0.25 \\ CPI = 1.25 \Rightarrow 20\% \text{ degradation} \end{array}}$$

Performance of Two-Level Unified Cache

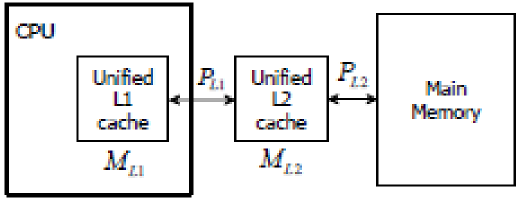




$$\text{CPI}^{\text{stall}} = \frac{\text{stall cycles}}{\text{IC}} = \sum_{n\,=\,\text{stall types}} \frac{\text{stall cycles}}{\text{stall type } n} \times \frac{\text{stalls of type } n}{\text{IC}}$$

$$\text{stall types} = \sum_{i\,=\,(\text{instruction, data})} \{(\text{L1 miss, L2 hit})_i, (\text{L1 miss, L2 miss})_i\}$$

Stalls in Two-Level Unified Cache



$$\mathrm{CPI}^{\text{stall}} = \sum_{i = \{\text{data, instr}\}} \left[\frac{\text{stall cycles}}{(\text{L1 miss, L2 hit})_i} \times \frac{(\text{L1 miss, L2 hit})_i}{\text{IC}} + \frac{\text{stall cycles}}{(\text{L1 miss, L2 miss})_i} \times \frac{(\text{L1 miss, L2 miss})_i}{\text{IC}} \right]$$

$$= \sum_{i = \{\text{data, instr}\}} \left[\frac{\text{stall cycles}}{(\text{L1 miss, L2 hit})_i} \times \frac{(\text{L1 miss, L2 hit})_i}{\text{L1 access}_i} \times \frac{\text{L1 access}_i}{\text{IC}} + \frac{\text{stall cycles}}{(\text{L1 miss, L2 miss})_i} \times \frac{(\text{L1 miss, L2 miss})_i}{\text{L1 access}_i} \times \frac{\text{L1 access}_i}{\text{IC}} \right]$$

Assume L1 and L2 Are Statistically Independent

$$\frac{(\text{L1 miss, L2 hit})_i}{\text{L1 access}_i} = \frac{\text{L1 miss}_i \text{ followed by L2 hit}}{\text{L1 access}_i} = \frac{\text{L1 miss}_i}{\text{L1 access}_i} \times \underbrace{\frac{\text{L2 access}}{\text{L1 miss}_i}}_{1} \times \frac{\text{L2 hit}}{\text{L2 access}}$$

$$= \frac{\text{L1 miss}_i}{\text{L1 access}_i} \times \frac{\text{L2 hit}}{\text{L2 access}}$$

$$\frac{(\text{L1 miss, L2 miss})_i}{\text{L1 access}_i} = \frac{\text{L1 miss}_i \text{ followed by L2 miss}}{\text{L1 access}_i} = \frac{\text{L1 miss}_i}{\text{L1 access}_i} \times \underbrace{\frac{\text{L2 access}}{\text{L1 miss}_i}}_{1} \times \frac{\text{L2 miss}}{\text{L2 access}}$$

$$= \frac{\text{L1 miss}_i}{\text{L1 access}_i} \times \frac{\text{L2 miss}}{\text{L2 access}}$$

$$\text{CPI}^{\text{stall}} = \sum_{i=\{\text{data, instr}\}} \left[\frac{\text{stall cycles}}{(\text{L1 miss, L2 hit})_i} \times \frac{\text{L1 miss}_i}{\text{L1 access}_i} \times \frac{\text{L2 hit}}{\text{L2 access}} \times \frac{\text{L1 access}_i}{\text{IC}} \right.$$
$$\left. + \frac{\text{stall cycles}}{(\text{L1 miss, L2 miss})_i} \times \frac{\text{L1 miss}_i}{\text{L1 access}_i} \times \frac{\text{L2 miss}}{\text{L2 access}} \times \frac{\text{L1 access}_i}{\text{IC}} \right]$$

Definitions

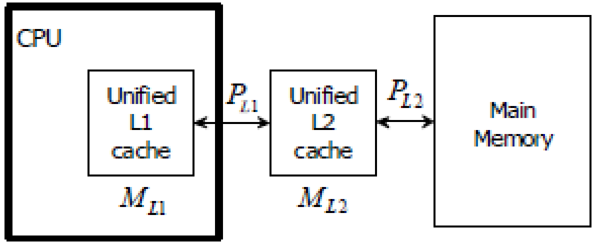
Miss Rates

$$M_{L1} = \text{Miss Rate at L1} = \frac{\text{L1 miss}}{\text{L1 access}}$$

$$1 - M_{L1} = \text{Hit Rate at L1} = \frac{\text{L1 hit}}{\text{L1 access}}$$

$$M_{L2} = \text{Miss Rate at L2} = \frac{\text{L2 miss}}{\text{L1 access}}$$

$$1 - M_{L2} = \text{Hit Rate at L2} = \frac{\text{L2 hit}}{\text{L2 access}}$$



Miss Penalties

$$P_{L1} = \text{Miss Penalty (cycles) from L1 to L2} = \frac{\text{stall cycles}}{\text{(L1 miss, L2 hit)}}$$

$$P_{L2} = \text{Miss Penalty (cycles) from L2 to Main Memory}$$

$$P_{L1} + P_{L2} = \frac{\text{stall cycles}}{\text{(L1 miss, L2 miss)}}$$

$$IC^{A} = \text{Data Access Instructions (Load/Store)}$$

$$IC = \text{Total Instructions}$$

Simplifying

$$\text{CPI}^{\text{stall}} = \sum_{i=\{\text{data, instr}\}} \left[\frac{\text{stall cycles}}{(\text{L1 miss, L2 hit})_i} \times \frac{\text{L1 miss}_i}{\text{L1 access}_i} \times \frac{\text{L2 hit}}{\text{L2 access}} \times \frac{\text{L1 access}_i}{\text{IC}} \right.$$

$$\left. + \frac{\text{stall cycles}}{(\text{L1 miss, L2 miss})_i} \times \frac{\text{L1 miss}_i}{\text{L1 access}_i} \times \frac{\text{L2 miss}}{\text{L2 access}} \times \frac{\text{L1 access}_i}{\text{IC}} \right]$$

$$= P_{L1} \times M_{L1} \times (1 - M_{L2}) \times \frac{IC + IC^A}{IC} + (P_{L1} + P_{L2}) \times M_{L1} \times M_{L2} \times \frac{IC + IC^A}{IC}$$

$$= M_{L1} \times \left[1 + \frac{IC^A}{IC}\right] \times [P_{L1} \times (1 - M_{L2}) + (P_{L1} + P_{L2}) \times M_{L2}]$$

$$= M_{L1} \times \left[1 + \frac{IC^A}{IC}\right] \times [P_{L1} - P_{L1} \times M_{L2} + P_{L1} \times M_{L2} + P_{L2} \times M_{L2}]$$

$$\text{CPI}^{\text{stall}} = M_{L1} \times \left[1 + \frac{IC^A}{IC}\right] \times [P_{L1} + P_{L2} \times M_{L2}]$$

Splitting L1 Cache with Unified L2 Cache

$$\frac{\text{misses at L1}}{\text{IC}} = \frac{\text{misses at L1}}{\text{accesses at L1}} \times \frac{\text{accesses at L1}}{\text{IC}}$$

$$= M_{L1} \times \left[1 + \frac{IC^A}{IC}\right] = M_{L1} + M_{L1} \times \frac{IC^A}{IC}$$

$$\downarrow$$

$$\frac{\text{data misses at L1}}{\text{data accesses at L1}} + \frac{\text{instruction misses at L1}}{\text{instruction accesses at L1}} = \sum_{i=(\text{data},\text{instructions})} \frac{\text{misses at L1}_i}{\text{accesses at L1}_i} \times \frac{\text{accesses at L1}_i}{\text{IC}}$$

$$= M_{L1}^{\text{instructions}} \times 1 + M_{L1}^{\text{data}} \times \frac{IC^A}{IC}$$

$$= M_{L1}^{\text{I}} + M_{L1}^{\text{D}} \times \frac{IC^A}{IC}$$

$$\text{CPI}^{\text{stall}}_{\text{unified}} = M_{L1} \times \left[1 + \frac{IC^A}{IC}\right] \times \left[P_{L1} + P_{L2} \times M_{L2}\right]$$

$$\downarrow$$

$$\text{CPI}^{\text{stall}}_{\text{split}} = \left[M_{L1}^{\text{I}} + M_{L1}^{\text{D}} \times \frac{IC^A}{IC}\right] \times \left[P_{L1} + P_{L2} \times M_{L2}\right]$$

Second Layer Cache (L2) with Split L1

Definitions

P_{L1} = Miss Penalty (cycles) at L1

P_{L2} = Miss Penalty (cycles) at L2

IC^{A} = Data Access Instructions (Load/Store)

IC = Total Instructions

M_{L1}^{I} = Instruction Miss Rate at L1

M_{L1}^{D} = Data Miss Rate at L1

M_{L2} = Miss Rate at L2

One layer (L1) of split cache

$$CPI_{1-level}^{stall} = \left[M_{L1}^{I} + M_{L1}^{D} \times \frac{IC^{A}}{IC} \right] \times P_{L1}$$

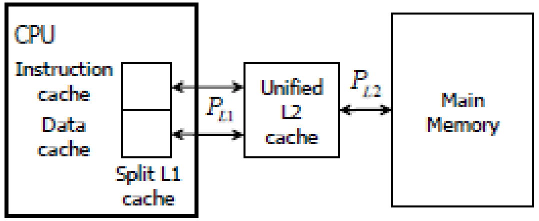
Miss Penalty at L1 (to main memory) P_{L1} ~ 50 cycles

Split L1 cache and unified L2 cache

Miss Penalty at L1 (to L2) P_{L1} ~ 5 cycles

Miss Penalty at L2 (to main memory) P_{L2} ~ 50 cycles

Miss Rate at L2 ~ 1%



$$CPI_{2-level}^{stall} = \left[M_{L1}^{I} + M_{L1}^{D} \times \frac{IC^{A}}{IC} \right] \times \left(P_{L1} + P_{L2} \times M_{L2} \right)$$

$$P_{L1} + P_{L2} \times M_{L2} = 5 + 50 \times 0.01 = 5.5$$

Miss Example — Extreme Locality

Program

```
int i,a;
for (i = 0 ; i < 4096 ; i++){
    a = i;
}
```

Compiler assignments

Register ← `i`

Memory ← `a`

Memory accesses

- 4096 write accesses to integer `a`
  - 1 compulsory cache miss (write allocate) on `i = 0`
  - 4095 cache hits on `i > 0`
- 0 read accesses to `a`

Miss rate

$$\text{miss rate} = \frac{1\ \text{miss}}{4096\ \text{accesses}} = 2.44\times10^{-4}$$

Miss Example — Extreme Non-Locality

Block size

16 bytes/block = 4 words/block = 4 array elements

每一块可以存放数组的4个元素

Program

```
int i,a[4096];
for (i = 0 ; i < 4096 ; i++){
     a[i] = i;
}
```

Compiler assignments

Register ← **i**

Memory ← **a[]**

Memory accesses

4096 write accesses to integer array **a[]**

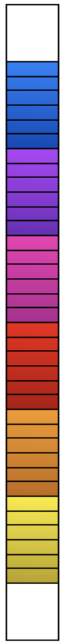
Compulsory cache miss every 4 array elements

4096/4 = 1024 cache misses (write allocate)

0 read accesses to **a[]**

Miss rate

$$\text{miss rate} = \frac{1024 \text{ misses}}{4096 \text{ accesses}} = 0.25$$



a[i][j]	j = 0	j = 1	j = 2	j = 3	j = 4	j = 5	j = 6	j = 7
i = 0	1 [m]	2 [h]	3 [h]	4 [h]	5 [m]	6 [h]	7 [h]	8 [h]
i = 1	9 [m]	10 [h]	11 [h]	12 [h]	13 [m]	14 [h]	15 [h]	16 [h]
i = 2	17 [m]	18 [h]	19 [h]	20 [h]	21 [m]	22 [h]	23 [h]	24 [h]
i = 3	25 [m]	26 [h]	27 [h]	28 [h]	29 [m]	30 [h]	31 [h]	32 [h]

Miss Example — Good Locality

Cache parameters	B	S	W	Capacity = B x S x W
	16	256	4	16 KB = 4 Kwords

4路组相连
每组256块
每块4个字（数组元素）（16字节）

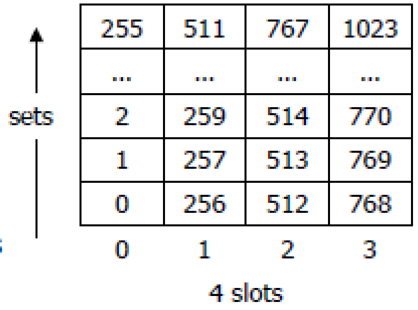
Program

```
int a[4096],i,j;
for (i = 0 ; i < 10 ; i++){
    for (j = 0 ; j < 4096 ; j++){
    a[j] = i + j;
    }
}
```

Compiler assignments

Register ← **i,j**

Memory ← **a[]**



0	1	2	3
255	511	767	1023
...	...	...	...
2	259	514	770
1	257	513	769
0	256	512	768

Memory accesses

40960 write accesses to integer array **a[]**

`i = 0:` Compulsory cache miss every 4 elements

`i > 0:` Entire array in cache ⇒ cache hits

4K integers = 16 Kbytes

= 1024 blocks

Miss rate

$$\text{miss rate} = \frac{1024 \text{ misses}}{40960 \text{ accesses}} = 0.025$$

Miss Example — Better Locality Using MRU (Most Recently Used)

Cache parameters	B	S	W	Capacity = B x S x W
	16	256	4	16 KB = 4 Kwords

4路组相连
每组256块
每块4个字（数组元素）（16字节）

Program

```
int a[5120],i,j;
for (i = 0 ; i < 10 ; i++){
    for (j = 0 ; j < 5120 ; j++){
    a[j] = i + j;
    }
}
```

Compiler assignments

Register ← **i,j**

Memory ← **a[]**

255	511	767	1023	
...	...	...	...	
2	259	514	770	
1	257	513	769	
0	256	512	768	
0	1	2	3	

4 slots

5K integers = 1280 blocks

Memory accesses with MRU

51200 write accesses to integer array **a[]**

`i = 0`: Compulsory cache miss every 4 elements

`i > 0`: Conflict misses to slot 3 (2 out of 5 accesses)

j=0...255, 1024...1279

Miss rate

$$\text{miss rate} = \frac{1280 + 512 \times 9 \text{ misses}}{51200 \text{ accesses}} = 0.025 \times \left(1 + \frac{2}{5} \times 9\right) = 0.115$$

Miss Example — Improved Locality

Cache parameters	B	S	W	Capacity = B x S x W
	16	256	4	16 KB = 4 Kwords

Compiler assignments

Register ← **i , j**

Memory ← **a[]**

Program

```
    int a[5120],i,j;
    for (i = 0 ; i < 10 ; i++){
        for (j = 0 ; j < 4096 ; j++){
        a[j] = i + j;
        }
    for (i = 0 ; i < 10 ; i++){
        for (j = 4096 ; j < 5120 ; j++){
        a[j] = i + j;
        }
    }
```

255	511	767	1023
...	...	...	...
2	259	514	770
1	257	513	769
0	256	512	768

1279	511	767	1023
...	...	...	...
1026	259	514	770
1025	257	513	769
1024	256	512	768

5K integers = 1280 blocks

Memory accesses

51200 write accesses to integer array **a[]**

`i = 0:` Compulsory cache miss every 4 elements

`i > 0:` Entire array in cache ⇒ cache hits

Miss rate

$$\text{miss rate} = \frac{1024 + 256 \text{ misses}}{40960 + 10240 \text{ accesses}} = 0.025$$

Miss Example — Address Aliasing

Cache parameters	B	S	W	Capacity = B x S x W
	16	256	2	8 KB = 2 Kwords

Program

```
int a[512],b[512],c[512],i,j;
 for (i = 0 ; i < 20; i++){
   for (j = 0 ; j < 512 ; j++){
     c[j] = a[j] + b[j] + c[j];
   }
 }
```

Compiler assignments

Register ← `i,j`

Memory (address = $0200AS_1S_2B$)

```
a: 02000000 – 020007FF
b: 02001000 – 020017FF
c: 02002000 – 020027FF
```

3 × 512 integers = 6 KB = 384 blocks

Memory accesses

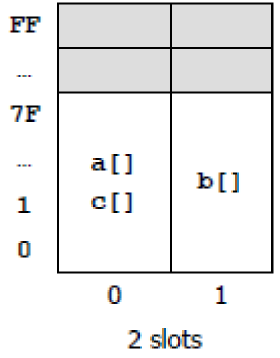
20 × 3 × 512 = 30720 read accesses to `a[], b[], c[]`

20 × 512 = 10240 write accesses to array `c[]`

Set assignment = $\text{int}(0200AS_1S_2B/10)\%100 = S_1S_2$

`i = 0:` 3 × 512 / 4 = 384 compulsory misses

`i > 0:` 2 × 512 / 4 = 256 conflict misses for sets `a[], c[]`



Miss rate

$$\text{miss rate} = \frac{384+256\times 19 \text{ misses}}{30720+10240 \text{ accesses}} = \frac{2688}{40960} = 0.128$$

Miss Example — Address Aliasing with Larger W

Cache parameters	B	S	W	Capacity = B x S x W
	16	128	4	8 KB = 2 Kwords

Program

```
int a[512],b[512],c[512],i,j;
for (i = 0 ; i < 20; i++){
  for (j = 0 ; j < 512 ; j++){
    c[j] = a[j] + b[j] + c[j];
  }
}
```

Compiler assignments

Register ← `i,j`

Memory (address = $0200AS_1S_2B$)

`a:`	`02000000`	–	`020007FF`
`b:`	`02001000`	–	`020017FF`
`c:`	`02002000`	–	`020027FF`

3 x 512 integers = 6 KB = 384 blocks

Memory accesses

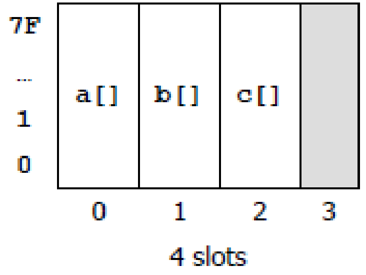
20 x 3 x 512 = 30720 read accesses to `a[], b[], c[]`

20 x 512 = 10240 write accesses to array `c[]`

Set assignment = `int(0200`AS_1S_2B`/10)%80` = S_1S_2

`i = 0:` 3 x 512 / 4 = 384 compulsory misses

`i > 0:` All arrays in cache ⇒ cache hits



Miss rate

$$\text{miss rate} = \frac{384 \text{ misses}}{30720+10240 \text{ accesses}} = \frac{384}{40960} = 0.009375$$

Writing Cache Friendly Code

Two major rules:

- **Repeated references to data are good (temporal locality)**
- **Stride reference patterns are good (spatial locality)**
- **Example: cold cache, 4-byte words, 4-word cache blocks**

```
int sum_array_rows(int a[M][N])
{
  int i, j, sum = 0;

  for (i = 0; i < M; i++)
      for (j = 0; j < N; j++)
          sum += a[i][j];
  return sum;
}
```

Miss rate = 1/4 = 25%

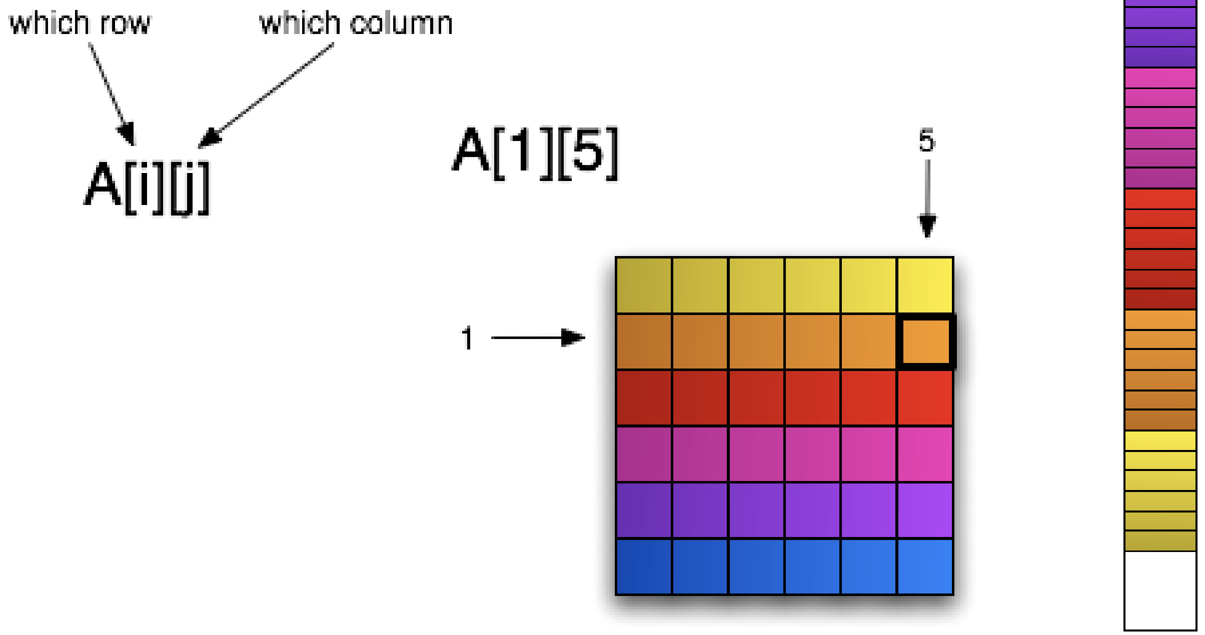
```
int sum_array_cols(int a[M][N])
{
  int i, j, sum = 0;

  for (j = 0; j < N; j++)
      for (i = 0; i < M; i++)
          sum += a[i][j];
  return sum;
}
```

Miss rate = 100%

Layout of C Arrays in Memory

C arrays allocated in **row-major order**

- each row in contiguous memory locations

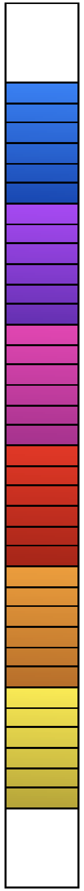


Layout of C Arrays in Memory

- C arrays allocated in row-major order
  - each row in contiguous memory locations
- Stepping through columns in one row:

  -
    ```
    for (i = 0; i < N; i++)
     sum += a[0][i];
    ```
  - accesses successive elements
  - if block size (B) > 4 bytes, exploit spatial locality
    - compulsory miss rate = 4 bytes / B
- Stepping through rows in one column:

  -
    ```
    for (i = 0; i < n; i++)
     sum += a[i][0];
    ```
  - accesses distant elements
  - no spatial locality!
    - compulsory miss rate = 1 (i.e. 100%)



Matrix Multiplication Example

Major Cache Effects to Consider

- **Total cache size**
  - Exploit temporal locality and keep the working set small (e.g., use blocking)
- **Block size**
  - Exploit spatial locality

Description:

- **Multiply N x N matrices**
- **O(N^3) total operations**
- **Accesses**
  - N reads per source element
  - N values summed per destination
    - but may be able to hold in regis

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

Variable *sum* held in register

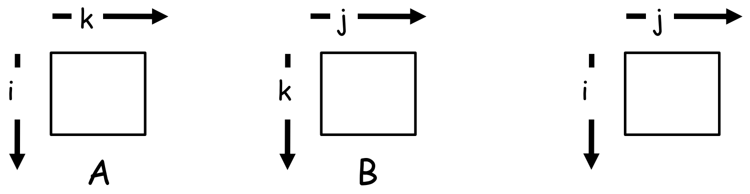
Miss Rate Analysis for Matrix Multiply

Assume:

- **Block size = 32B (big enough for four 64-bit words)**
- **Matrix dimension (N) is very large**
  - Approximate 1/N as 0.0
- **Cache is not even big enough to hold multiple rows**

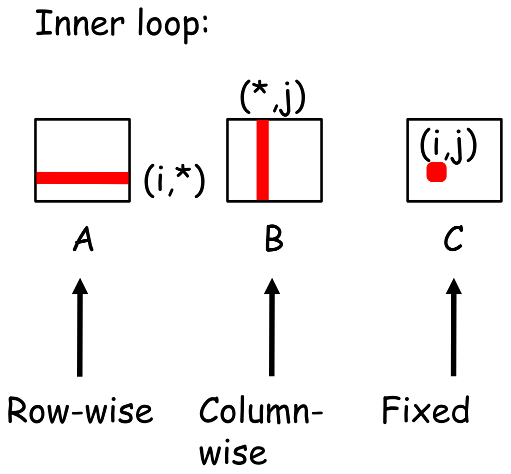
Analysis Method:

- **Look at access pattern of inner loop**



Matrix Multiplication (ijk)

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

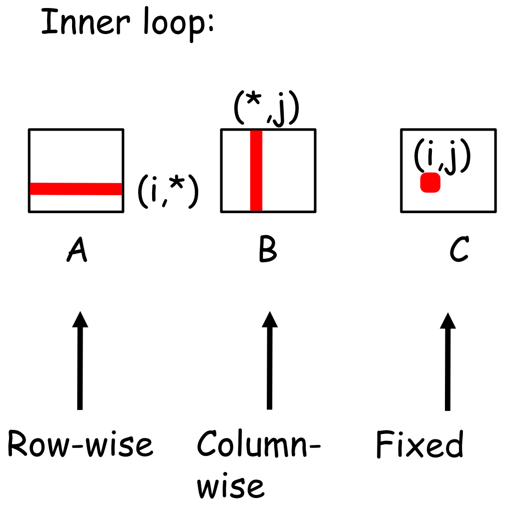


Misses per Inner Loop Iteration:

A	B	C
0.25	1.0	0.0 (=1/N)

Matrix Multiplication (jik)

```
/* jik */
for (j=0; j<n; j++) {
  for (i=0; i<n; i++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum
  }
}
```

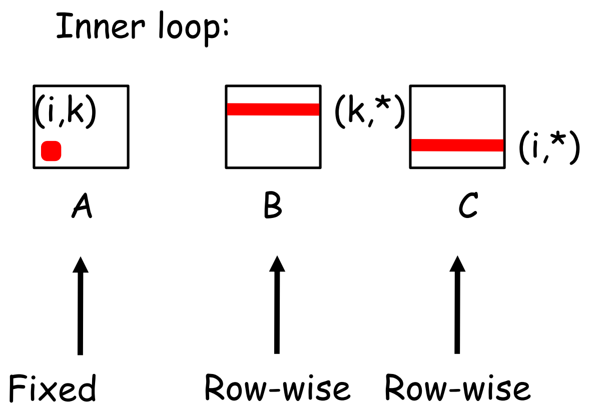


Misses per Inner Loop Iteration:

A	B	C
0.25	1.0	0.0

Matrix Multiplication (kij)

```
/* kij */
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

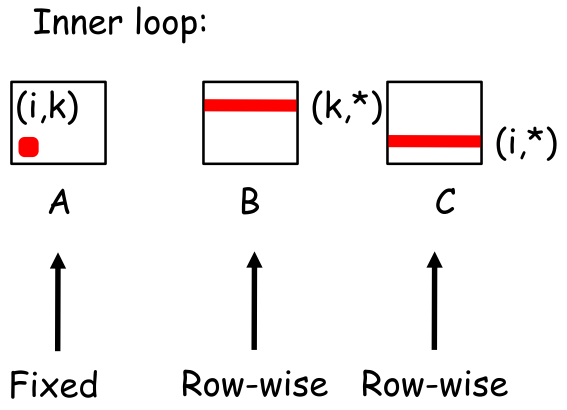


Misses per Inner Loop Iteration:

A	B	C
0.0	0.25	0.25

Matrix Multiplication (ikj)

```
/* ikj */
for (i=0; i<n; i++) {
  for (k=0; k<n; k++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```



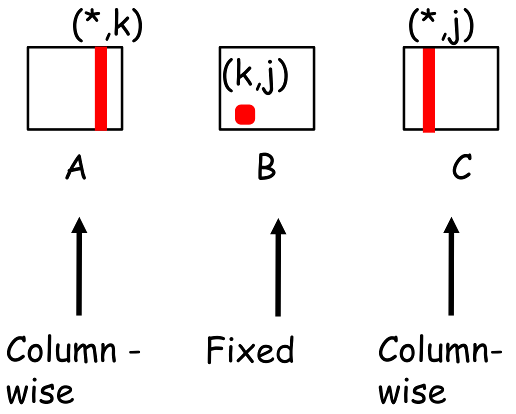
Misses per Inner Loop Iteration:

A	B	C
0.0	0.25	0.25

Matrix Multiplication (jki)

```
/* jki */
for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

Inner loop:

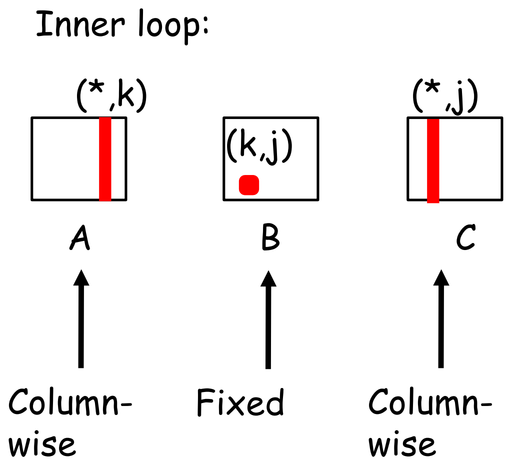


Misses per Inner Loop Iteration:

A	B	C
1.0	0.0	1.0

Matrix Multiplication (kji)

```
/* kji */
for (k=0; k<n; k++) {
  for (j=0; j<n; j++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```



Misses per Inner Loop Iteration:

A	B	C
1.0	0.0	1.0

Summary of Matrix Multiplication

```
 for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
   sum = 0.0;
   for (k=0; k<n; k++)
     sum += a[i][k] * b[k][j];
   c[i][j] = sum;
 }
}
```

ijk (& jik):
- 2 loads, 0 stores
- misses/iter = 1.25

```
for (k=0; k<n; k++) {
 for (i=0; i<n; i++) {
  r = a[i][k];
  for (j=0; j<n; j++)
   c[i][j] += r * b[k][j];
 }
}
```

kij (& ikj):
- 2 loads, 1 store
- misses/iter = 0.5

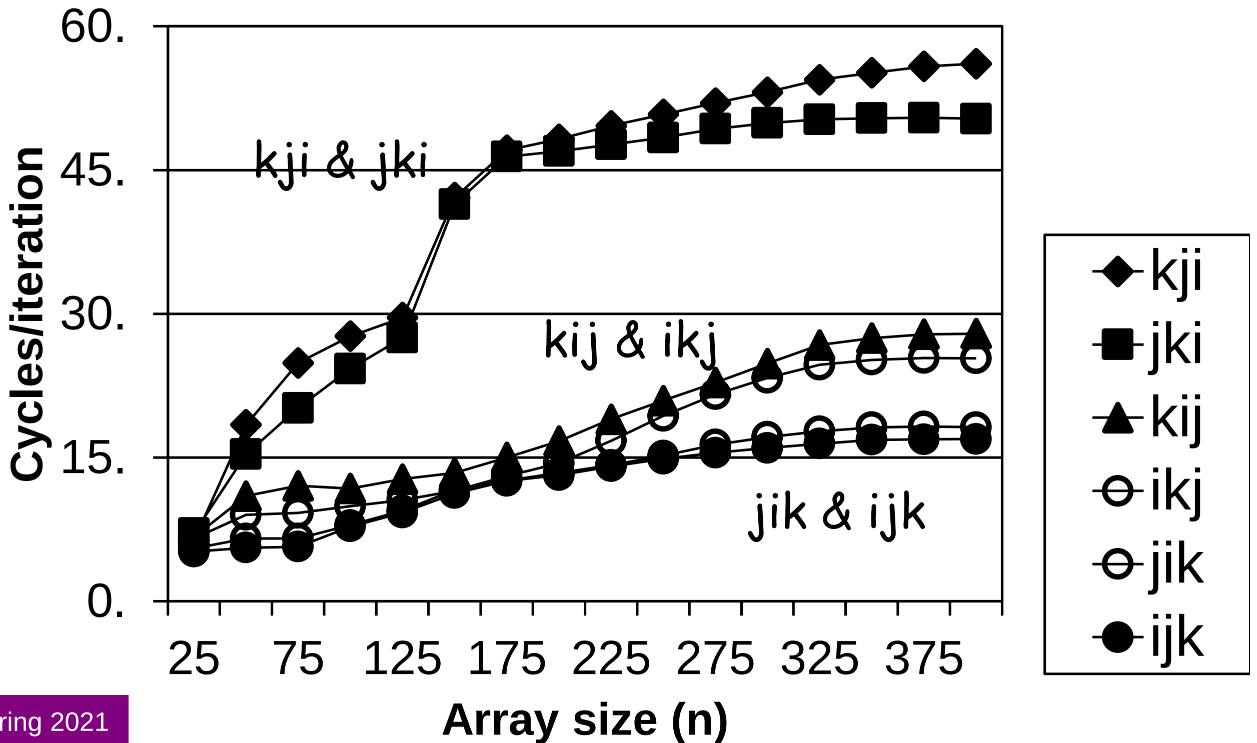
```
for (j=0; j<n; j++) {
 for (k=0; k<n; k++) {
   r = b[k][j];
   for (i=0; i<n; i++)
    c[i][j] += a[i][k] * r;
 }
}
```

jki (& kji):
- 2 loads, 1 store
- misses/iter = 2.0

Pentium Matrix Multiply Performance

Miss rates are helpful but not perfect predictors.

- Code scheduling matters, too.



Improving Temporal Locality by Blocking

Example: Blocked matrix multiplication

- **"block" (in this context) does not mean "cache block".**
- **Instead, it mean a sub-block within the matrix.**
- **Example: N = 8; sub-block size = 4**

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

Key idea: Sub-blocks (i.e., A_{xy}) can be treated just like scalars.

$$C_{11} = A_{11}B_{11} + A_{12}B_{21} \qquad C_{12} = A_{11}B_{12} + A_{12}B_{22}$$

$$C_{21} = A_{21}B_{11} + A_{22}B_{21} \qquad C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

Blocked Matrix Multiply (bijk)

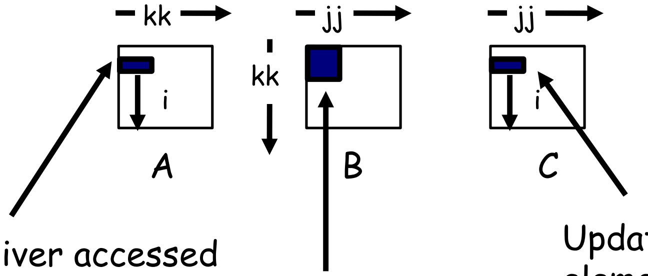
```
for (jj=0; jj<n; jj+=bsize) {

  for (i=0; i<n; i++)
    for (j=jj; j < min(jj+bsize,n); j++)
      c[i][j] = 0.0;

  for (kk=0; kk<n; kk+=bsize) {
    for (i=0; i<n; i++) {
      for (j=jj; j < min(jj+bsize,n); j++) {
        sum = 0.0
        for (k=kk; k < min(kk+bsize,n); k++) {
          sum += a[i][k] * b[k][j];
        }
        c[i][j] += sum;
      }
    }
  }
}
```

Blocked Matrix Multiply Analysis

- **Innermost loop pair multiplies a *1 X bsize* sliver of *A* by a *bsize X bsize* block of *B* and accumulates into *1 X bsize* sliver of *C***
- **Loop over *i* steps through *n* row slivers of *A* & *C*, using same *B***

```
for (i=0; i<n; i++) {
    for (j=jj; j < min(jj+bsize,n); j++) {
      sum = 0.0
      for (k=kk; k < min(kk+bsize,n); k++) {
        sum += a[i][k] * b[k][j];
      }
      c[i][j] += sum;
    }
```

Innermost Loop Pair



row sliver accessed bsize times

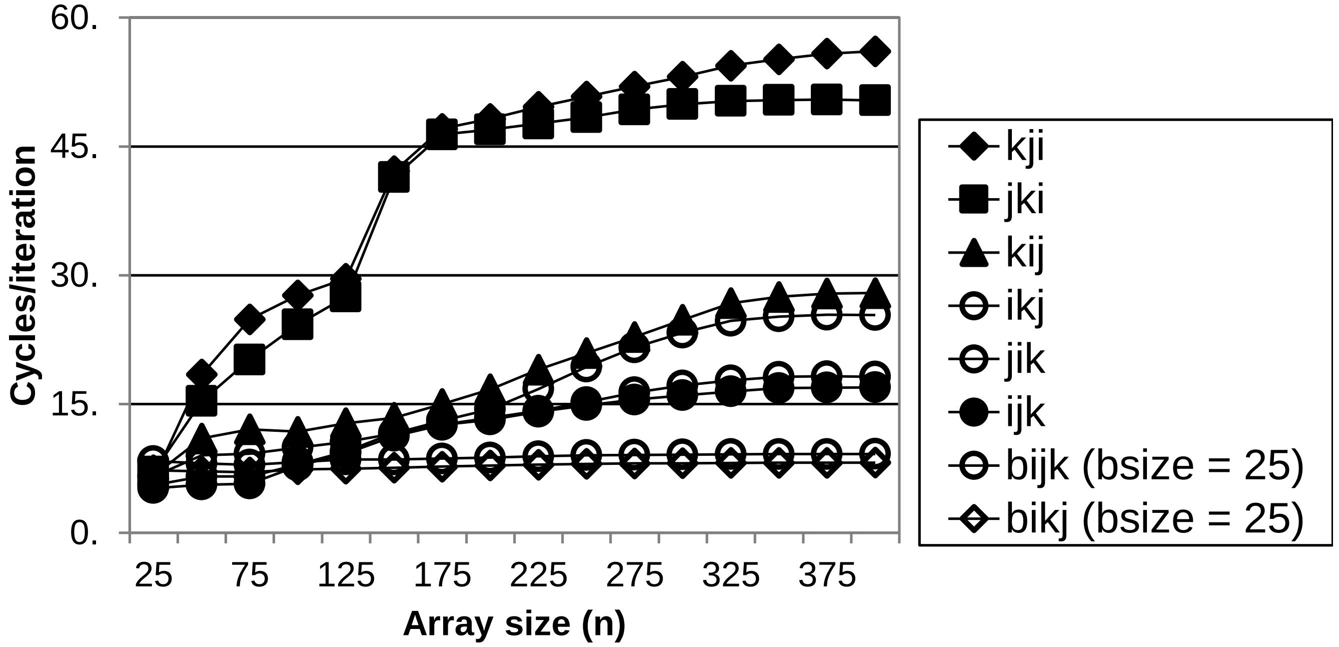
block reused n times in succession

Update successive elements of sliver

Pentium Blocked Matrix Multiply Performance

Blocking (bijk and bikj) improves performance by a factor of two over unblocked versions (ijk and jik)

- **relatively insensitive to array size.**



Concluding Observations

Programmer can optimize for cache performance

- How data structures are organized
- How data are accessed
  - Nested loop structure
  - Blocking is a general technique

All systems favor "cache friendly code"

- Getting absolute optimum performance is very platform specific
  - Cache sizes, line sizes, associativities, etc.
- Can get most of the advantage with generic code
  - Keep working set reasonably small (temporal locality)
  - Use small strides (spatial locality)