Memory Hierarchy --- Caching

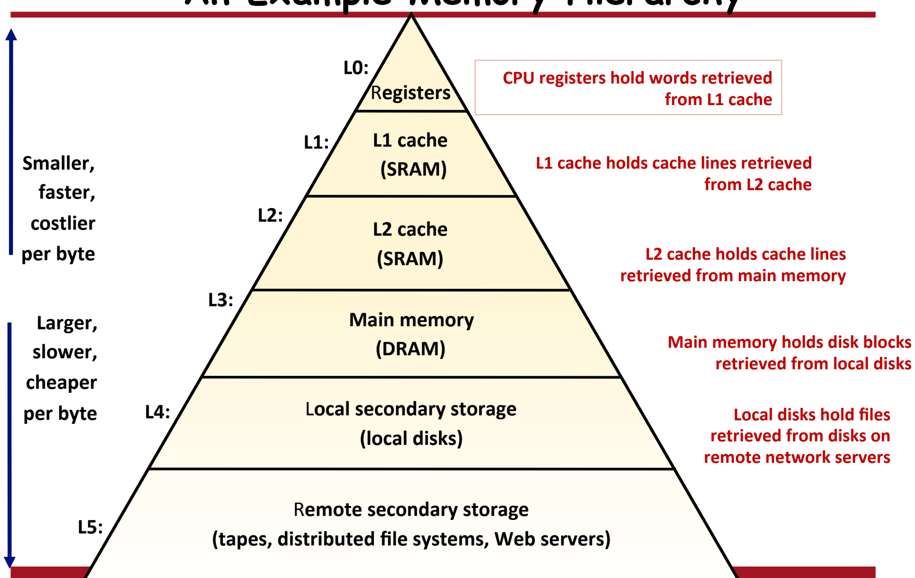
CSCI 2021: Machine Architecture and Organization

Antonia Zhai
Department Computer Science and Engineering
University of Minnesota
http://www.cs.umn.edu/~zhai

With Slides from Bryant and O'Hallaron

UNIVERSITY OF MINNESOTA

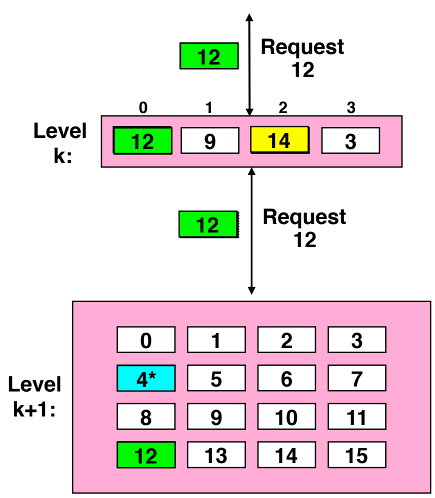
An Example Memory Hierarchy



Caches

- *Cache:* A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device.
- Fundamental idea of a memory hierarchy:
  - For each k, the faster, smaller device at level k serves as a cache for the larger, slower device at level k+1.
- Why do memory hierarchies work?
  - Because of locality, programs tend to access the data at level k more often than they access the data at level k+1.
  - Thus, the storage at level k+1 can be slower, and thus larger and cheaper per bit.
- *Big Idea:* The memory hierarchy creates a large pool of storage that costs as much as the cheap storage near the bottom, but that serves data to programs at the rate of the fast storage near the top.

General Caching Concepts



Program needs object d, which is stored in some block b.

Cache hit

- Program finds b in the cache at level k. E.g., block 14.

Cache miss

- b is not at level k, so level k cache must fetch it from level k+1. E.g., block 12.
- If level k cache is full, then some current block must be replaced (evicted). Which one is the "victim"?
  - Placement policy: where can the new block go? E.g., b mod 4
  - Replacement policy: which block should be evicted? E.g., LRU

General Caching Concepts: Types of Cache Misses

- Cold (compulsory) miss
  - Cold misses occur because the cache is empty.
- Conflict miss
  - Most caches limit blocks at level k+1 to a small subset (sometimes a singleton) of the block positions at level k.
    - E.g. Block i at level k+1 must be placed in block (i mod 4) at level k.
  - Conflict misses occur when the level k cache is large enough, but multiple data objects all map to the same level k block.
    - E.g. Referencing blocks 0, 8, 0, 8, 0, 8, ... would miss every time.
- Capacity miss
  - Occurs when the set of active cache blocks (working set) is larger than the cache.

Locality

Principle of Locality:

- Temporal locality: Recently referenced items are likely to be referenced in the near future.
- Spatial locality: Items with nearby addresses tend to be referenced close together in time.

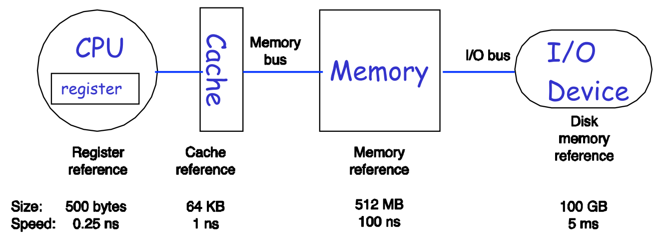
```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

Locality Example:

- **Data**
  - Reference array elements in succession **Spatial locality**
  - Reference `sum` each iteration: **Temporal locality**
- **Instructions**
  - Reference instructions in sequence: **Spatial locality**
  - Cycle through loop repeatedly: **Temporal locality**

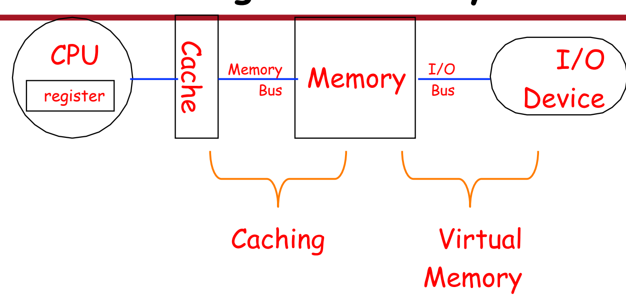
Storage Units Organization

- Rule-of-Thumb: The larger the structure, the slower the access time
- Solution: Hierarchical design



- There is locality in data access
- Put things that you are likely to use in the near future close to you

Caching the Memory

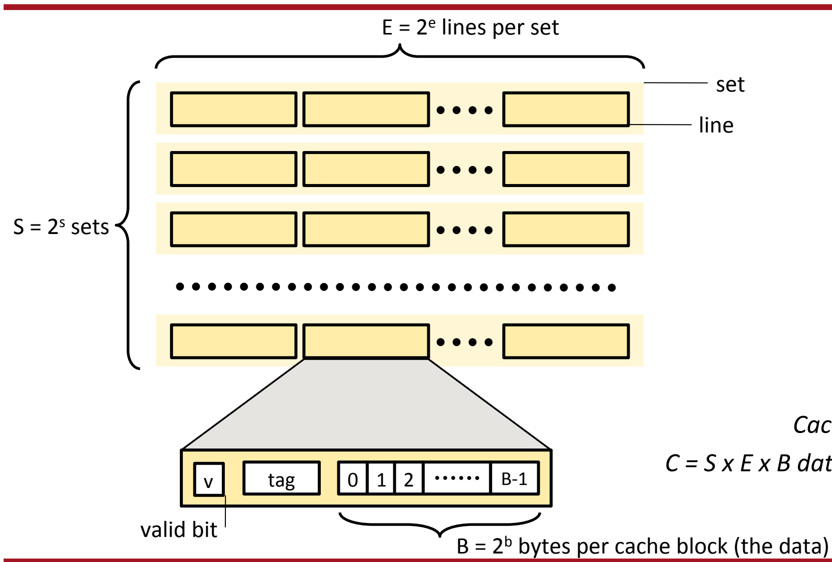


- Cache uses *SRAM*: Static Random Access Memory
  - No refresh
- Main Memory is *DRAM*: Dynamic Random Access Memory
  - Dynamic since needs to be refreshed periodically

The 1,2,3,4 of Caching

1. Where can a block be placed in the upper level?
2. How is a block found if it is in the upper level?
3. Which block should be replaced on a miss?
4. What happens on a write?

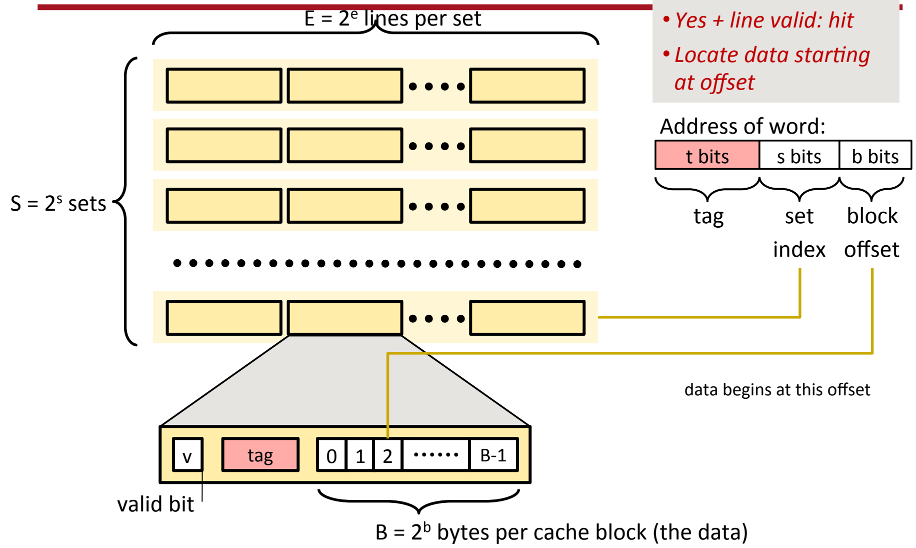
General Cache Organization (S, E, B)



Cache size:
C = S x E x B data bytes

Cache Read

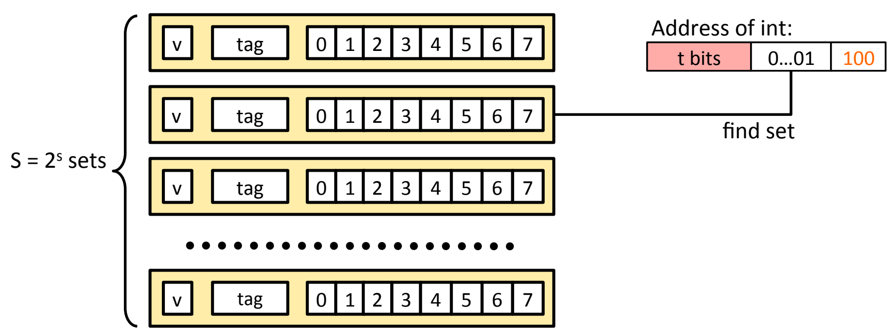
- *Locate set*
- *Check if any line in set has matching tag*
- *Yes + line valid: hit*
- *Locate data starting at offset*



Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set

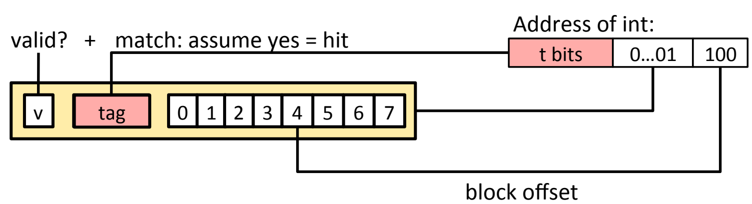
Assume: cache block size 8 bytes



Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set

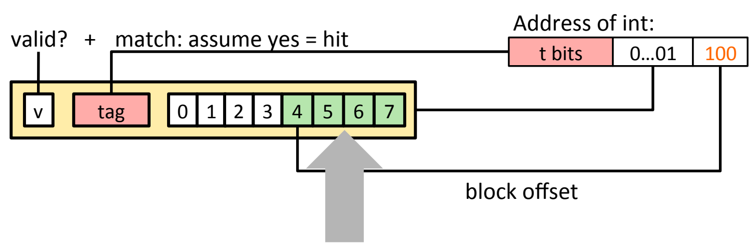
Assume: cache block size 8 bytes



Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set

Assume: cache block size 8 bytes



No match: old line is evicted and replaced

Direct-Mapped Cache Simulation

t=1	s=2	b=1
x	xx	x

M=16 byte addresses, B=2 bytes/block,
S=4 sets, E=1 Blocks/set

Address trace (reads, one byte per read):

0	$[0\underline{00}0_2]$,	
1	$[0\underline{00}1_2]$,	miss
7	$[0\underline{11}1_2]$,	hit
8	$[1\underline{00}0_2]$,	miss
0	$[0\underline{00}0_2]$	miss
		miss

	v	Tag	Block
Set 0	1	0	M[0-1]
Set 1			
Set 2			
Set 3	1	0	M[6-7]

Example

P

c
a
c
h
e

Data	Tag

Data	Tag

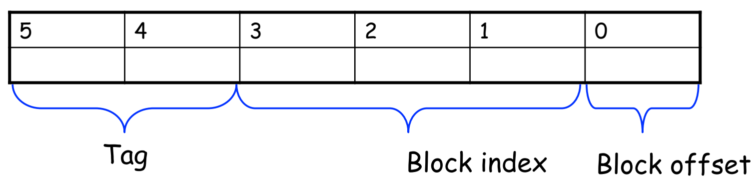
Cache block: 2 bytes
Associativity: Directly Mapped
Cache size: 16 bytes

m
e
m
o
r
y

	000	001	010	011	100	101	110	111
000	3	5	7	9	11	13	15	17
001	19	21	23	25	27	29	31	33
010	35	37	39	41	43	45	47	49
011	51	53	55	57	59	61	63	65
100	67	69	71	73	75	77	79	81
101	83	85	87	89	91	93	95	97
110	99	101	103	105	107	109	111	113
111	115	117	119	121	123	125	127	129

64 bytes

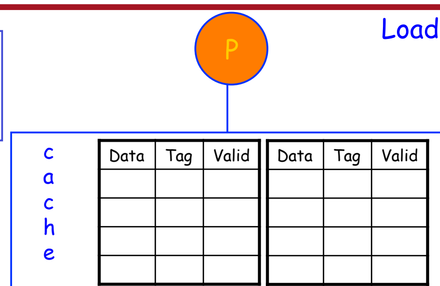
16



Example

```
for(I = 0; I < 100; i++)  {
    … = a[i];
}
```

Compulsory miss
Capacity miss



memory

	000	001	010	011	100	101	110	111
000	3	5	7	9	11	13	15	17
001	19	21	23	25	27	29	31	33
010	35	37	39	41	43	45	47	49
011	51	53	55	57	59	61	63	65
100	67	69	71	73	75	77	79	81
101	83	85	87	89	91	93	95	97
110	99	101	103	105	107	109	111	113
111	115	117	119	121	123	125	127	129

Load the following

010000
010001
010010
010011
010100
010101
010110
010111
011000
011001
011010
011011
011100
011101
011110
011111
10000

Example

```
for(I = 0; I < 100; i++)  {
    … = a[i];
}
```

Compulsory miss
Capacity miss

memory

	000	001	010	011	100	101	110	111
000	3	5	7	9	11	13	15	17
001	19	21	23	25	27	29	31	33
010	35	37	39	41	43	45	47	49
011	51	53	55	57	59	61	63	65
100	67	69	71	73	75	77	79	81
101	83	85	87	89	91	93	95	97
110	99	101	103	105	107	109	111	113
111	115	117	119	121	123	125	127	129

cache

Data	Tag	Valid	Data	Tag	Valid

Load the following

010000
010001
010010
010011
010100
010101
010110
010111
011000
011001
011010
011011
011100
011101
011110
011111
10000

19

Example

```
for(I = 0; I < 100; i++)  {
    … = a[i];
    … = b[i];
}
```

Conflict miss

P

cache

Data	Tag	Valid	Data	Tag	Valid

memory

	000	001	010	011	100	101	110	111
000	3	5	7	9	11	13	15	17
001	19	21	23	25	27	29	31	33
010	35	37	39	41	43	45	47	49
011	51	53	55	57	59	61	63	65
100	67	69	71	73	75	77	79	81
101	83	85	87	89	91	93	95	97
110	99	101	103	105	107	109	111	113
111	115	117	119	121	123	125	127	129

Load the following

010000
110000
010001
110001
010010
110010

20

Example

for(I = 0; I < 100; i++) {
 … = a[i];
 … = b[i];
}

Load the following

010000
110000
010001
110001
010010
110010

memory

	000	001	010	011	100	101	110	111
000	3	5	7	9	11	13	15	17
001	19	21	23	25	27	29	31	33
010	35	37	39	41	43	45	47	49
011	51	53	55	57	59	61	63	65
100	67	69	71	73	75	77	79	81
101	83	85	87	89	91	93	95	97
110	99	101	103	105	107	109	111	113
111	115	117	119	121	123	125	127	129

Conflict miss

cache

Data	Tag	Valid	Data	Tag	Valid

Example

for(I = 0; I < 100; i++) {
 … = a[i];
 … = b[i];
 … = c[i];
}

P

Load the following

010000
110000
111000
010001
110001
111001
010010
110010
111010

cache

Data	Tag	Valid	Data	Tag	Valid

Conflict miss

memory

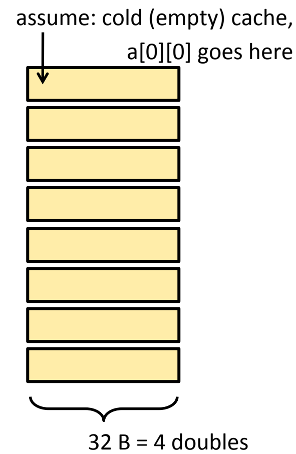
	000	001	010	011	100	101	110	111
000	3	5	7	9	11	13	15	17
001	19	21	23	25	27	29	31	33
010	35	37	39	41	43	45	47	49
011	51	53	55	57	59	61	63	65
100	67	69	71	73	75	77	79	81
101	83	85	87	89	91	93	95	97
110	99	101	103	105	107	109	111	113
111	115	117	119	121	123	125	127	129

Ignore the variables sum, i, j

A Higher Level Example

```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

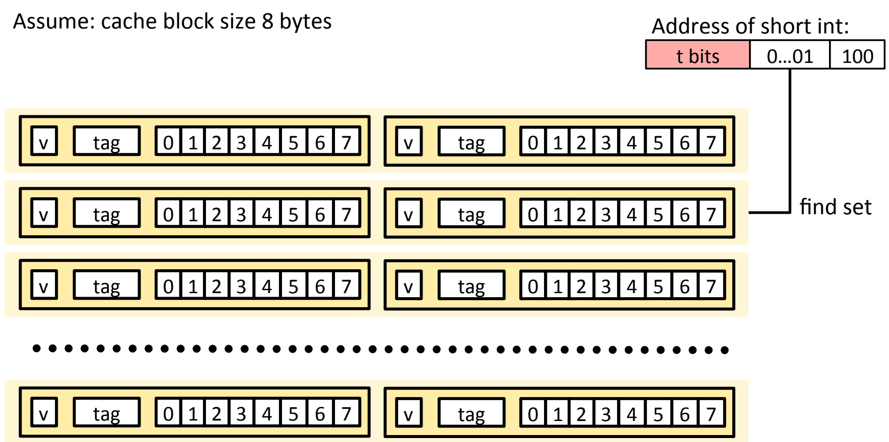
```
int sum_array_cols(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; i < 16; i++)
        for (i = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```



E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set

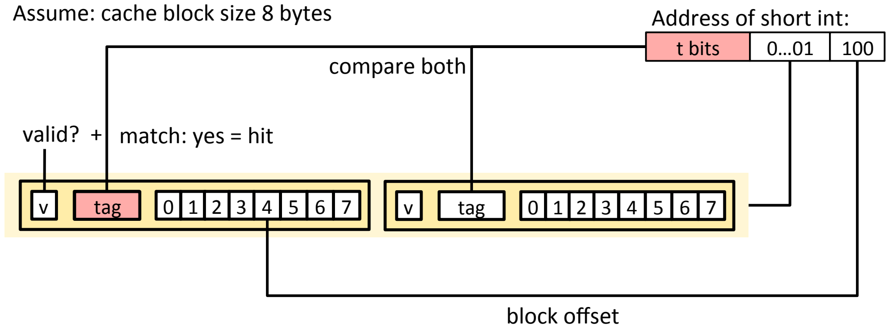
Assume: cache block size 8 bytes



E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set

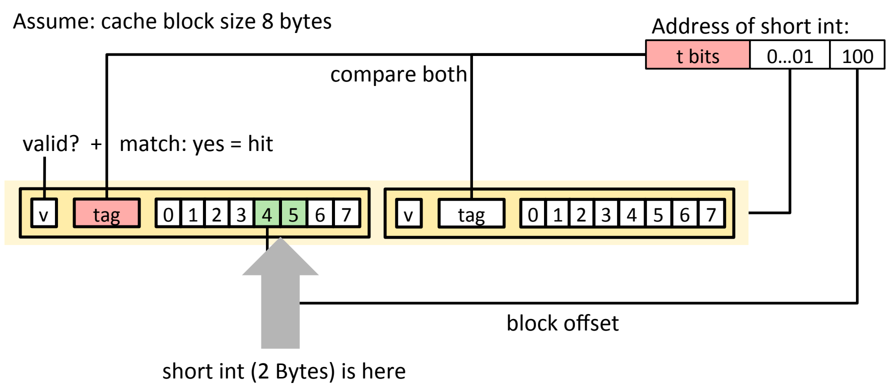
Assume: cache block size 8 bytes



E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set

Assume: cache block size 8 bytes



No match:

- One line in set is selected for eviction and replacement
- Replacement policies: random, least recently used (LRU), …

2-Way Set Associative Cache Simulation

t=2	s=1	b=1
xx	x	x

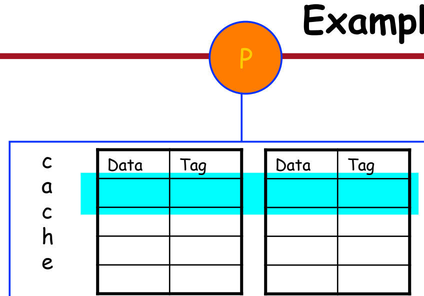
M=16 byte addresses, B=2 bytes/block,
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

0	$[00\underline{0}0_2]$,	miss
1	$[00\underline{0}1_2]$,	hit
7	$[01\underline{1}1_2]$,	miss
8	$[10\underline{0}0_2]$,	miss
0	$[00\underline{0}0_2]$	hit

	v	Tag	Block
Set 0	1	00	M[0-1]
	1	10	M[8-9]
Set 1	1	01	M[6-7]
	0		

Example



Cache block: 2 bytes
Associativity: 2
Cache size: 16 bytes

memory

	000	001	010	011	100	101	110	111
000	3	5	7	9	11	13	15	17
001	19	21	23	25	27	29	31	33
010	35	37	39	41	43	45	47	49
011	51	53	55	57	59	61	63	65
100	67	69	71	73	75	77	79	81
101	83	85	87	89	91	93	95	97
110	99	101	103	105	107	109	111	113
111	115	117	119	121	123	125	127	129

64 bytes

28

5	4	3	2	1	0

Tag — Block index — Block offset

Example

```
for(I = 0; I < 100; i++)  {
    … = a[i];
}
```

Compulsory miss
Capacity miss

P

cache

Data	Tag	Valid	Data	Tag	Valid

memory

	000	001	010	011	100	101	110	111
000	3	5	7	9	11	13	15	17
001	19	21	23	25	27	29	31	33
010	35	37	39	41	43	45	47	49
011	51	53	55	57	59	61	63	65
100	67	69	71	73	75	77	79	81
101	83	85	87	89	91	93	95	97
110	99	101	103	105	107	109	111	113
111	115	117	119	121	123	125	127	129

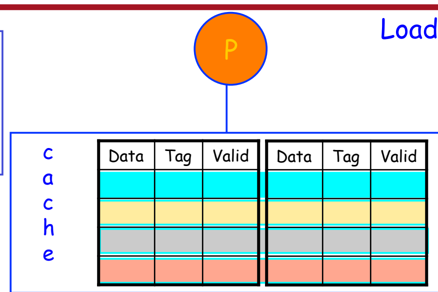
Load the following

010000
010001
010010
010011
010100
010101
010110
010111
011000
011001
011010
011011
011100
011101
011110
011111
10000

Example

```
for(I = 0; I < 100; i++)  {
    … = a[i];
    … = b[i];
}
```

Conflict miss



Load the following

010000
110000
010001
110001
010010
110010

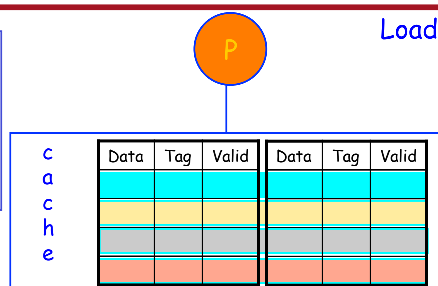
memory

	000	001	010	011	100	101	110	111
000	3	5	7	9	11	13	15	17
001	19	21	23	25	27	29	31	33
010	35	37	39	41	43	45	47	49
011	51	53	55	57	59	61	63	65
100	67	69	71	73	75	77	79	81
101	83	85	87	89	91	93	95	97
110	99	101	103	105	107	109	111	113
111	115	117	119	121	123	125	127	129

Example

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

Conflict miss



Load the following

010000
110000
111000
010001
110001
111001
010010
110010
111010

memory

	000	001	010	011	100	101	110	111
000	3	5	7	9	11	13	15	17
001	19	21	23	25	27	29	31	33
010	35	37	39	41	43	45	47	49
011	51	53	55	57	59	61	63	65
100	67	69	71	73	75	77	79	81
101	83	85	87	89	91	93	95	97
110	99	101	103	105	107	109	111	113
111	115	117	119	121	123	125	127	129

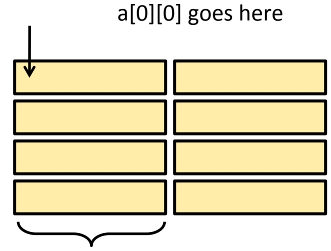
Ignore the variables sum, i, j

A Higher Level Example

```
int sum_array_rows(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (i = 0; i < 16; i++)
        for (j = 0; j < 16; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sum_array_col(double a[16][16])
{
    int i, j;
    double sum = 0;

    for (j = 0; j < 16; j++)
        for (i = 0; i < 16; i++)
            sum += a[i][j];
    return sum;
}
```

assume: cold (empty) cache,
a[0][0] goes here

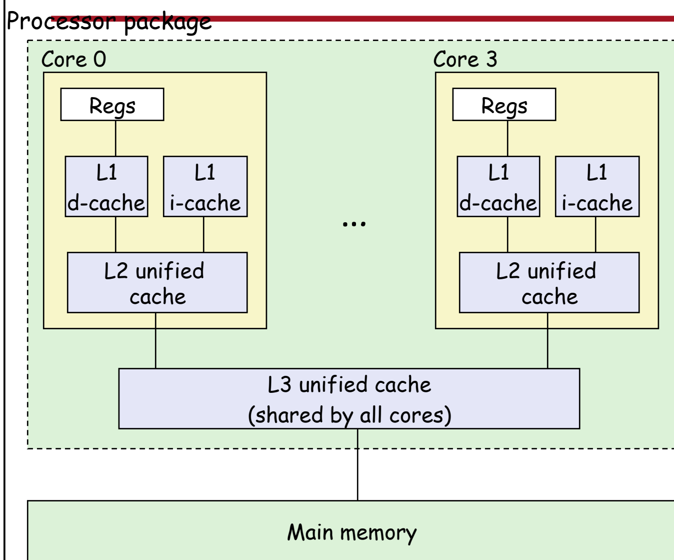


32 B = 4 doubles

What about writes?

- Multiple copies of data exist:
  - L1, L2, Main Memory, Disk
- What to do on a write-hit?
  - Write-through (write immediately to memory)
  - Write-back (defer write to memory until replacement of line)
    - Need a dirty bit (line different from memory or not)
- What to do on a write-miss?
  - Write-allocate (load into cache, update line in cache)
    - Good if more writes to the location follow
  - No-write-allocate (writes immediately to memory)
- Typical
  - Write-through + No-write-allocate
  - **Write-back + Write-allocate**

Intel Core i7 Cache Hierarchy



L1 i-cache and d-cache:
32 KB, 8-way,
Access: 4 cycles

L2 unified cache:
256 KB, 8-way,
Access: 11 cycles

L3 unified cache:
8 MB, 16-way,
Access: 30-40 cycles

Block size: 64 bytes for all caches.

Cache Performance

Cache Performance Metrics

Miss Rate

- Fraction of memory references not found in cache (misses/references)
- Typical numbers:
  - 3-10% for L1; can be quite small for L2, depending on size, etc.

Hit Time

- Time to deliver a line in the cache to the processor
  - includes time to determine whether the line is in the cache
- Typical numbers:
  - 1 clock cycle for L1; 3-8 clock cycles for L2

Miss Penalty

- Additional time required because of a miss
  - Typically 25-100 cycles for accessing the main memory

Average Memory Access Time

AMAT = Average Memory Access Time

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

- Repeated references to variables are good (temporal locality)
- Stride-1 reference patterns are good (spatial locality)

Lets think about those numbers

- Huge difference between a hit and a miss
  - Could be 100x, if just L1 and main memory

- Would you believe 99% hits is twice as good as 97%?
  - Consider:
    cache hit time of 1 cycle
    miss penalty of 100 cycles

  - Average access time:
    97% hits: 1 cycle + 0.03 * 100 cycles = **4 cycles**
    99% hits: 1 cycle + 0.01 * 100 cycles = **2 cycles**

- This is why "miss rate" is used instead of "hit rate"

Writing Cache Friendly Code

- Make the common case go fast
  - Focus on the inner loops of the core functions

- Minimize the misses in the inner loops
  - Repeated references to variables are good (temporal locality)
  - Stride-1 reference patterns are good (spatial locality)

Key idea: Our qualitative notion of locality is quantified through our understanding of cache memories.

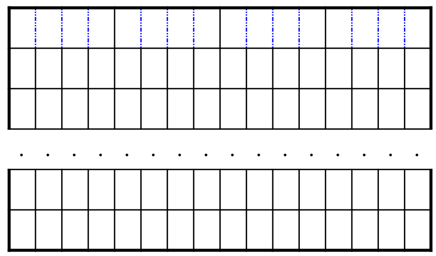
Writing Cache Friendly Code

16-bit address space

```
for (i = 0; i < 997; i++)
    for (j = 0; j < 997; j++)
        sum += a[i][j];
return sum;
```

Direct-mapped, 4-byte words, 4-word cache blocks, 1024 bytes in size

Cache



Writing Cache Friendly Code

0x0000
0x0004
0x0008
0x000C
0x0010
0x0014
0x0018
0x001C

0x0000
0x0F94
0x1F28
0x2EBC
. . .
0x0004
0x0F98
0x1F2C

```
for (i = 0; i < 997; i++)
    for (j = 0; j < 997; j++)
        sum += a[i][j];
return sum;
```

Miss rate = 1/4 = 25%

```
for (j = 0; j < 997; j++)
    for (i = 0; i < 997; i++)
        sum += a[i][j];
return sum;
```

Miss rate = 100%

How to Improve Cache Performance?

Hit Time = 1 Cycle

Miss Penalty = 100 Cycle

$$\textit{Average memory access time} = \textit{HitTime} + \textit{MissRate} \times \textit{MissPenalty}$$

1. Reduce the miss rate,
2. Reduce the miss penalty, or
3. Reduce the time to hit in the cache.

What if we have a bigger cache?

Another Example: Matrix Multiplication

Description:

- Multiply N x N matrices
- O(N3) total operations

Accesses

- N reads per source element
- N values summed per destination

```
/* 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

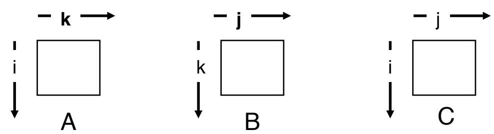
Miss Rate Analysis for Matrix Multiply

Assume:

- Line size = 32B (big enough for 4 8-byte words)
- Matrix dimension (N) is very large
- Cache is not even big enough to hold multiple rows

Analysis Method:

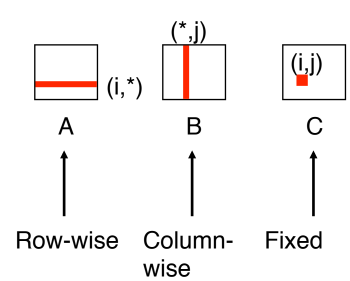
- Look at access pattern of inner loop



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;
  }
}
```

Inner loop:



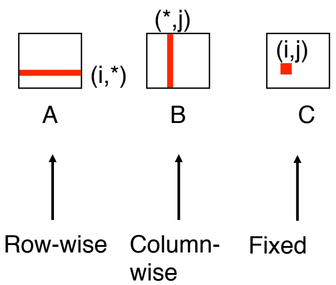
- <u>Misses per Inner Loop Iteration:</u>

<u>A</u>	<u>B</u>	<u>C</u>
0.25	1.0	0.0

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
  }
}
```

Inner loop:



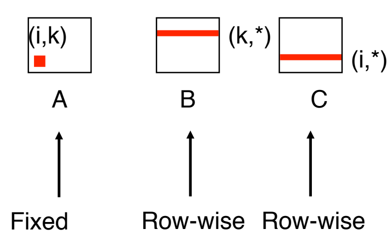
Misses per Inner Loop Iteration:

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];
  }
}
```

Inner loop:



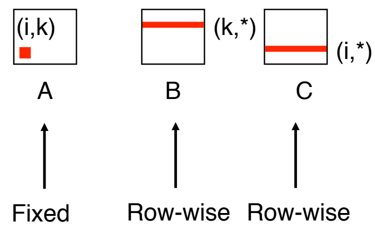
- Misses per Inner Loop Iteration:

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];
  }
}
```

Inner loop:



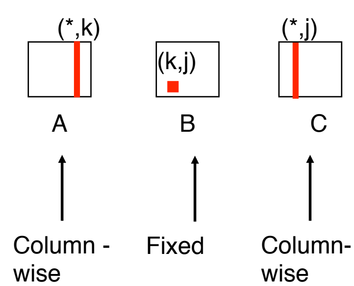
- Misses per Inner Loop Iteration:

<u>A</u>	<u>B</u>	<u>C</u>
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:

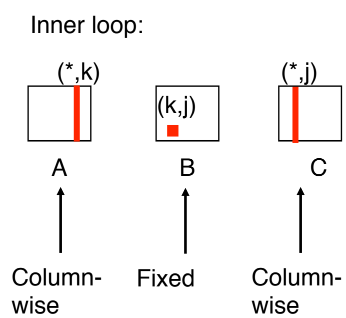


- Misses per Inner Loop Iteration:

<u>A</u>	<u>B</u>	<u>C</u>
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

Cache hit = 1 cycle; Cache miss penalty = 100 cycle

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

```
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;
  }
}
```

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

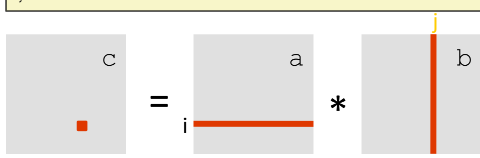
```
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

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

```
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;
  }
}
```

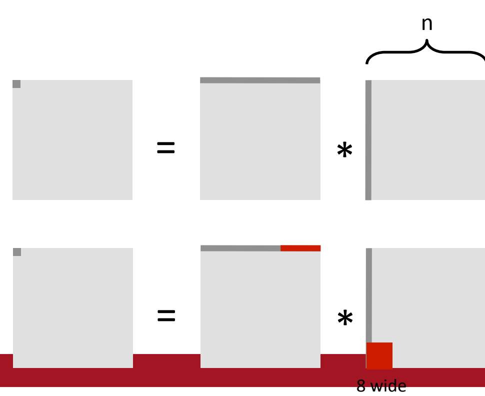
Example: Matrix Multiplication

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
            c[i*n+j] += a[i*n + k]*b[k*n + j];
}
```



Cache Miss Analysis

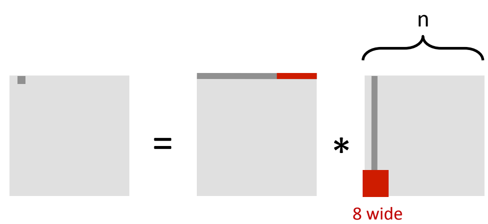
- Assume:
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)
- First iteration:
  - n/8 + n = 9n/8 misses
  - Afterwards in cache: (schematic)



Cache Miss Analysis

- Assume:
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)

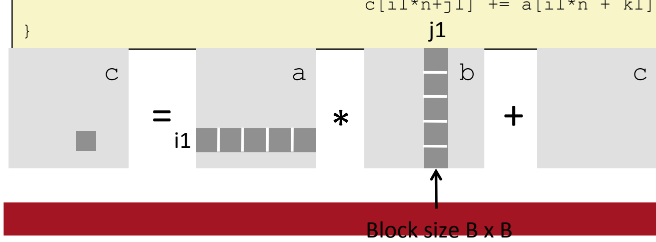
- Second iteration:
  - Again:
    n/8 + n = 9n/8 misses



- Total misses:
  - $9n/8 * n^2 = (9/8) * n^3$

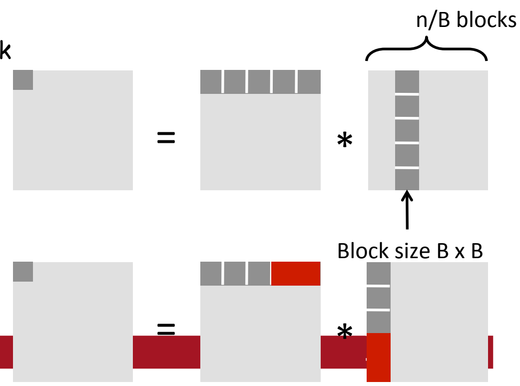
Blocked Matrix Multiplication

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i+=B)
        for (j = 0; j < n; j+=B)
            for (k = 0; k < n; k+=B)
                /* B x B mini matrix multiplications */
                for (i1 = i; i1 < i+B; i++)
                    for (j1 = j; j1 < j+B; j++)
                        for (k1 = k; k1 < k+B; k++)
                            c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
```



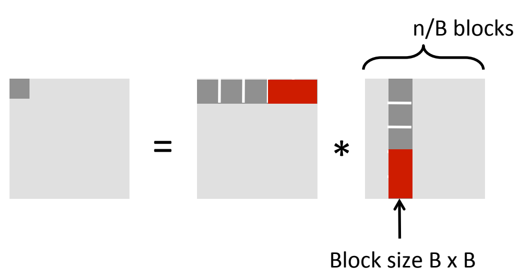
Cache Miss Analysis

- Assume:
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)
  - Three blocks fit into cache: $3B^2 < C$
- First (block) iteration:
  - $B^2/8$ misses for each block
  - $2n/B * B^2/8 = nB/4$ (omitting matrix c)
  - Afterwards in cache (schematic)



Cache Miss Analysis

- Assume:
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)
  - Three blocks fit into cache: $3B^2 < C$
- Second (block) iteration:
  - Same as first iteration
  - $2n/B * B^2/8 = nB/4$
- Total misses:
  - $nB/4 * (n/B)^2 = n^3/(4B)$



Summary

- No blocking: (9/8) * n^3
- Blocking: 1/(4B) * n^3

- Suggest largest possible block size B, but limit $3B^2 < C$!

- Reason for dramatic difference:
  - Matrix multiplication has inherent temporal locality:
    - Input data: $3n^2$, computation $2n^3$
    - Every array elements used O(n) times!
  - But program has to be written properly

Concluding Observations

Programmer can optimize for cache performance

- How data structures are organized
- How data are accessed
  - Nested loop structure

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)