Memory Hierarchy II

Last class

Caches

- Direct mapped
  - E=1 (One cache line per set)
  - Each main memory address can be placed in exactly one place in the cache
  - Conflict misses if two addresses map to same place in direct-mapped cache

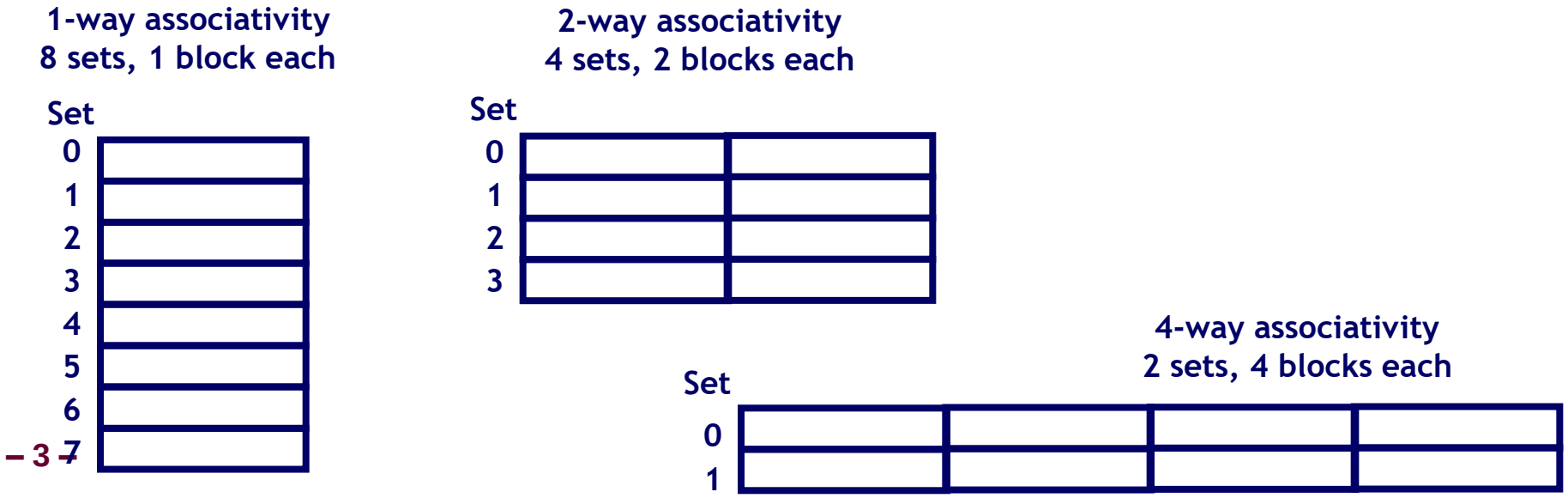
Set associativity

Set associative caches

- Reduce conflict misses
- Each set can store multiple distinct blocks/lines
- Each memory address maps to exactly one set in the cache, but data may be placed in any block within that set.

If each set has 2^x blocks, the cache is an 2^x-way associative cache.

Here are several possible organizations of an eight-block cache.



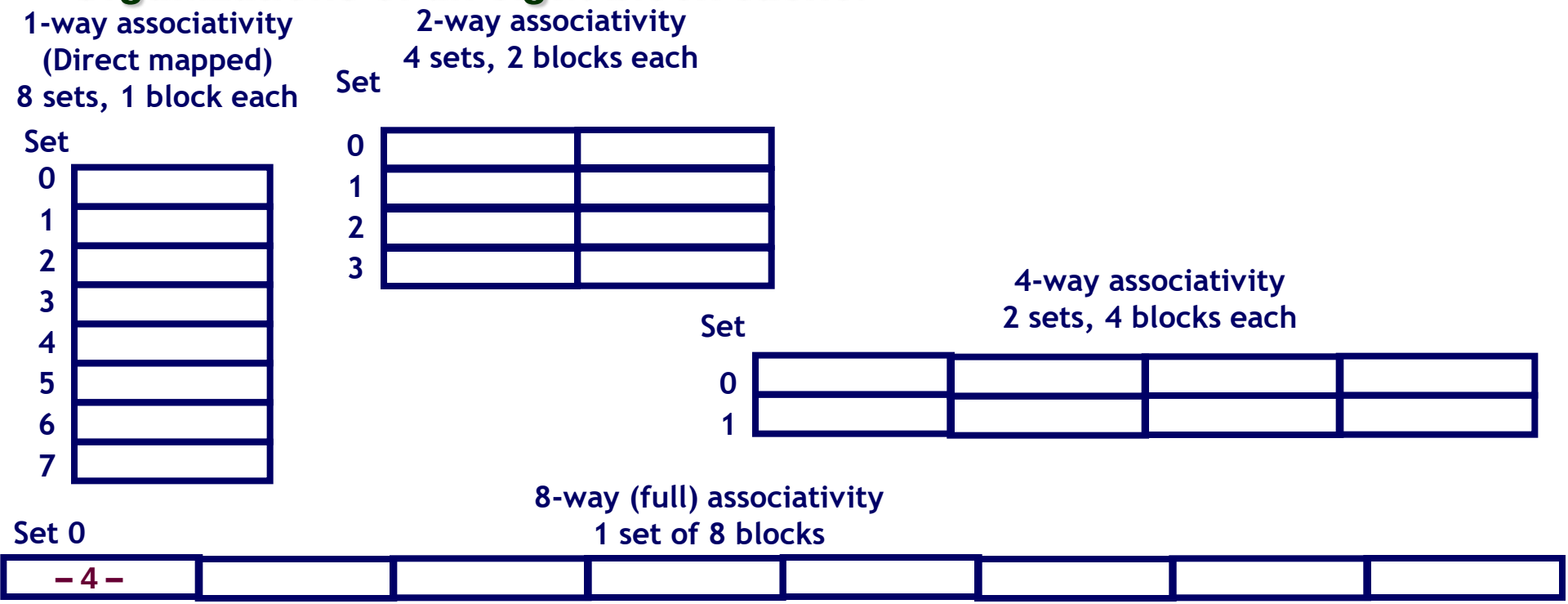
Set associative caches

Cache is divided into *groups* of blocks, called sets.

- **Each memory address maps to exactly one set in the cache, but data may be placed in any block within that set.**

If each set has 2^x blocks (E), the cache is an 2^x-way associative cache.

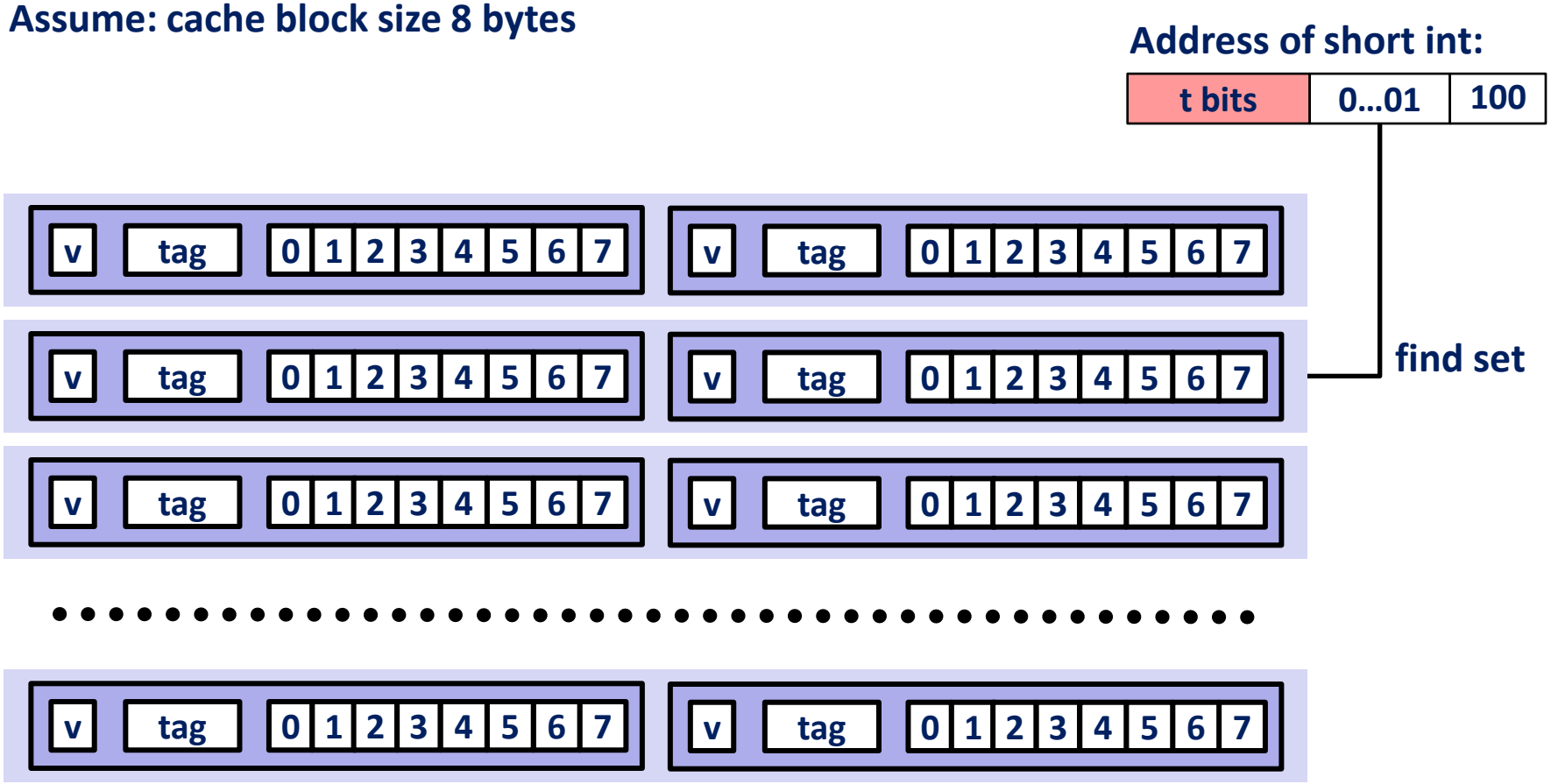
Organizations of an eight-block cache.



E-way Set Associative Cache (E = 2)

E = 2: Two lines per set

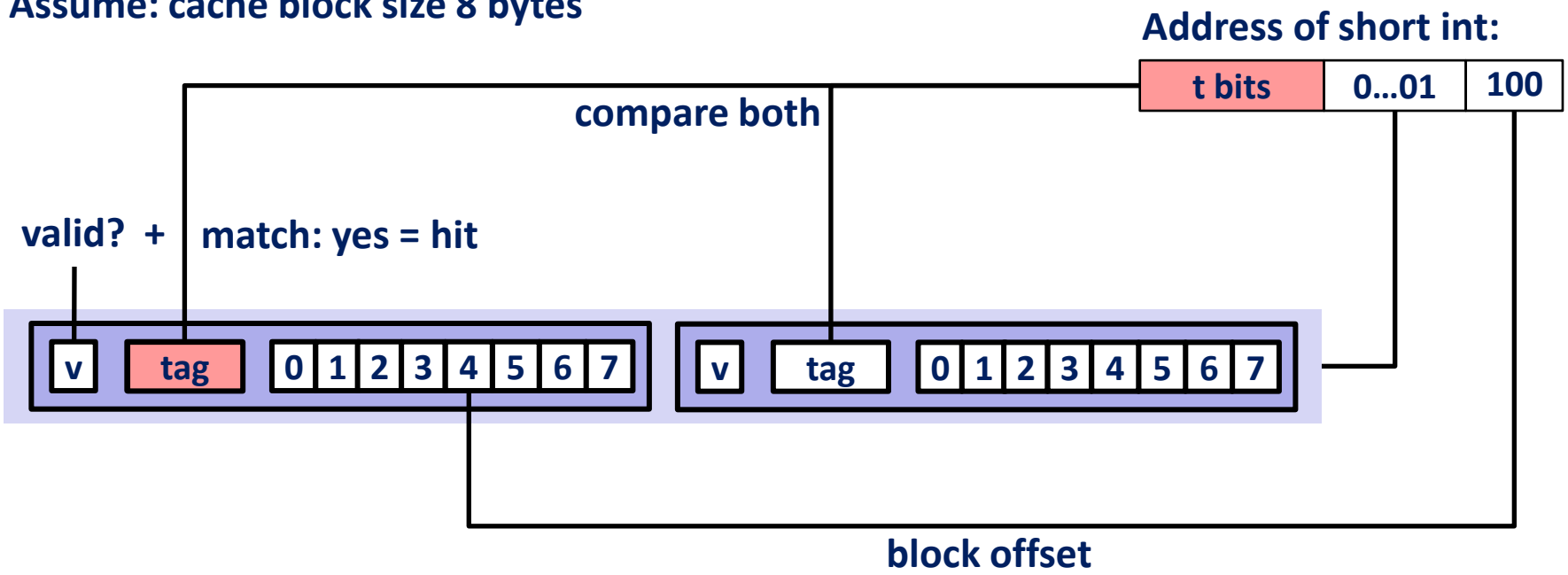
Assume: cache block size 8 bytes



E-way Set Associative Cache (E = 2)

E = 2: Two lines per set

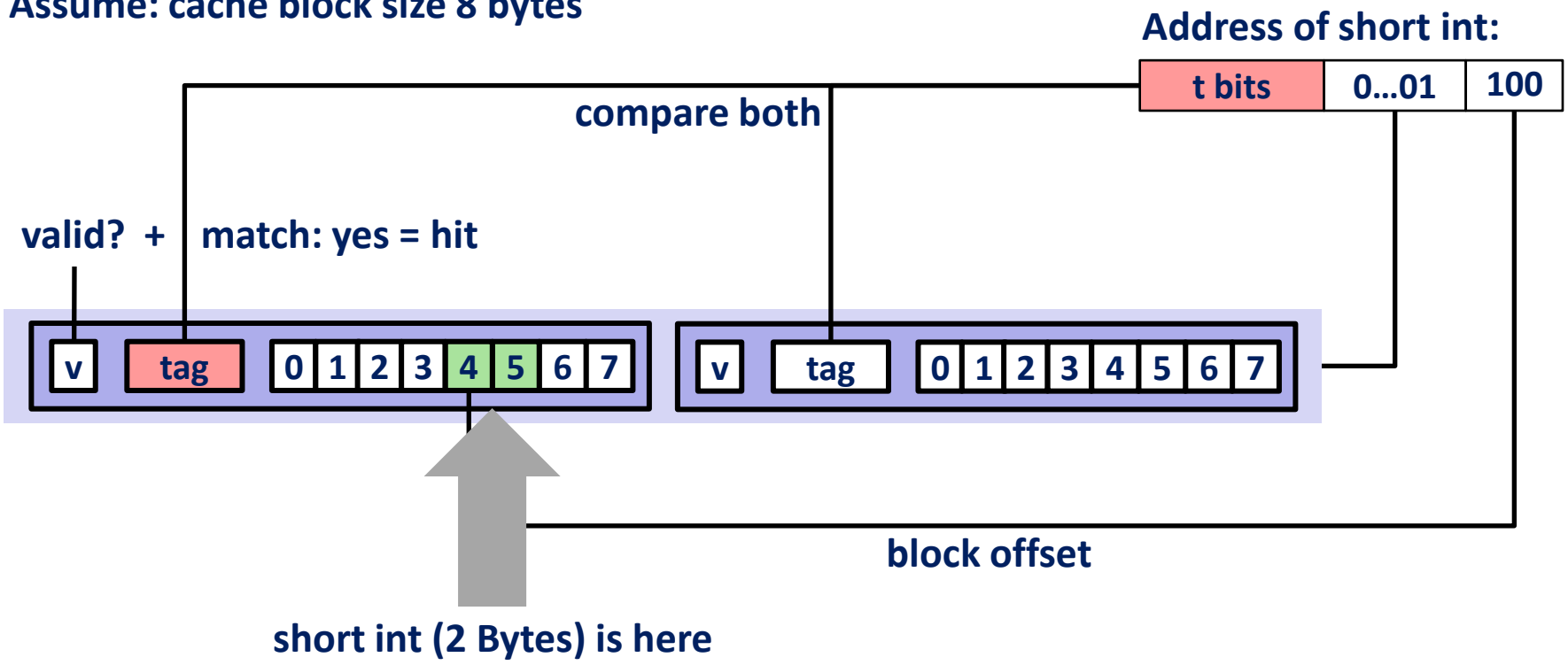
Assume: cache block size 8 bytes



E-way Set Associative Cache (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):

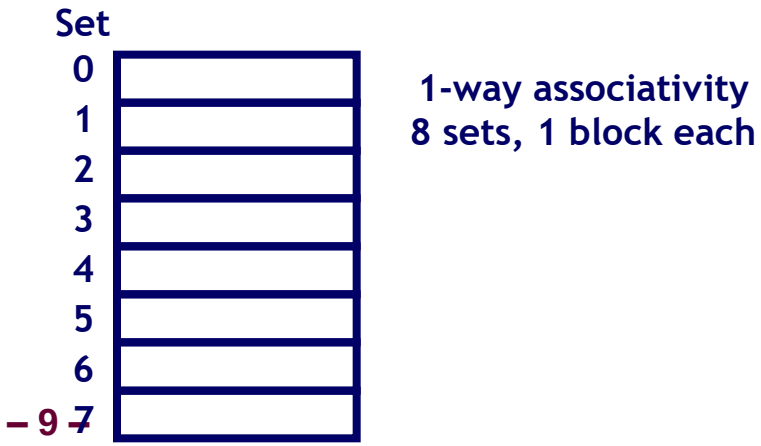
0	[00$\underline{0}$0$_2$],	miss
1	[00$\underline{0}$1$_2$],	hit
7	[01$\underline{1}$1$_2$],	miss
8	[10$\underline{0}$0$_2$],	miss
0	[00$\underline{0}$0$_2$]	hit

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

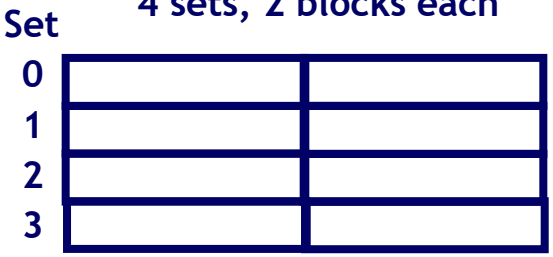
Examples

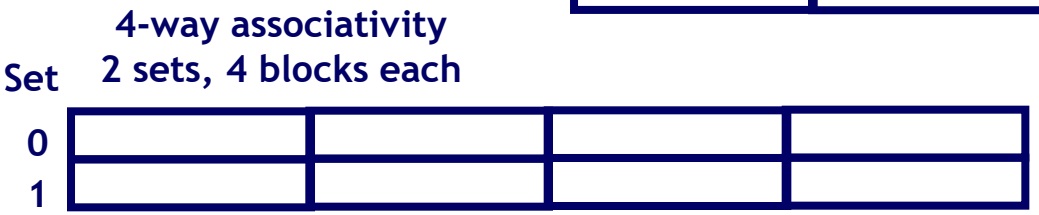
Assume 8 block cache with 16 bytes per block

- Draw a 1-way (direct mapped) implementation
- Draw a 2-way associative implementation
- Draw a 4-way associative implementation



1-way associativity
8 sets, 1 block each





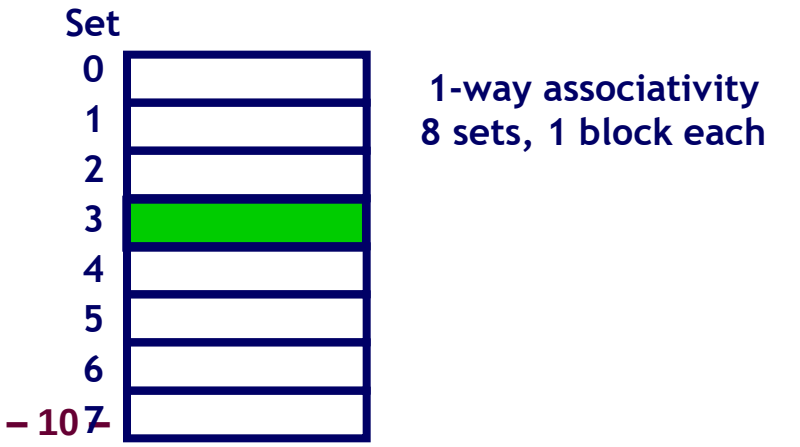
Examples

Assume 8 block cache with 16 bytes per block

- Where would data from this address go 1100000110011
- B=16 bytes, so the lowest 4 bits are the block offset: 110000011 0011
- Set index
  - 1-way (i.e. direct mapped) cache
    - » 8 sets so next three bits (011) are the set index. 110000 011 0011.
  - 2-way cache
    - » 4 sets so next two bits (11) are the set index. 1100000 11 0011.
  - 4-way cache
    - » 2 sets so the next one bit (1) is the set index. 11000001 1 0011.

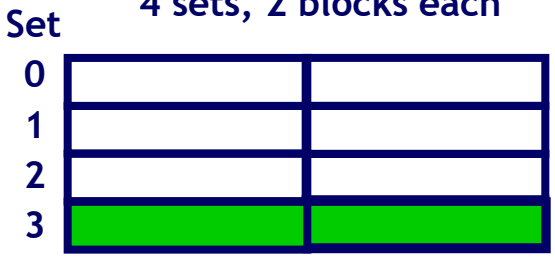
Data may go in *any* block (in green) within the correct set.

- Top bits are always for the tag



1-way associativity
8 sets, 1 block each

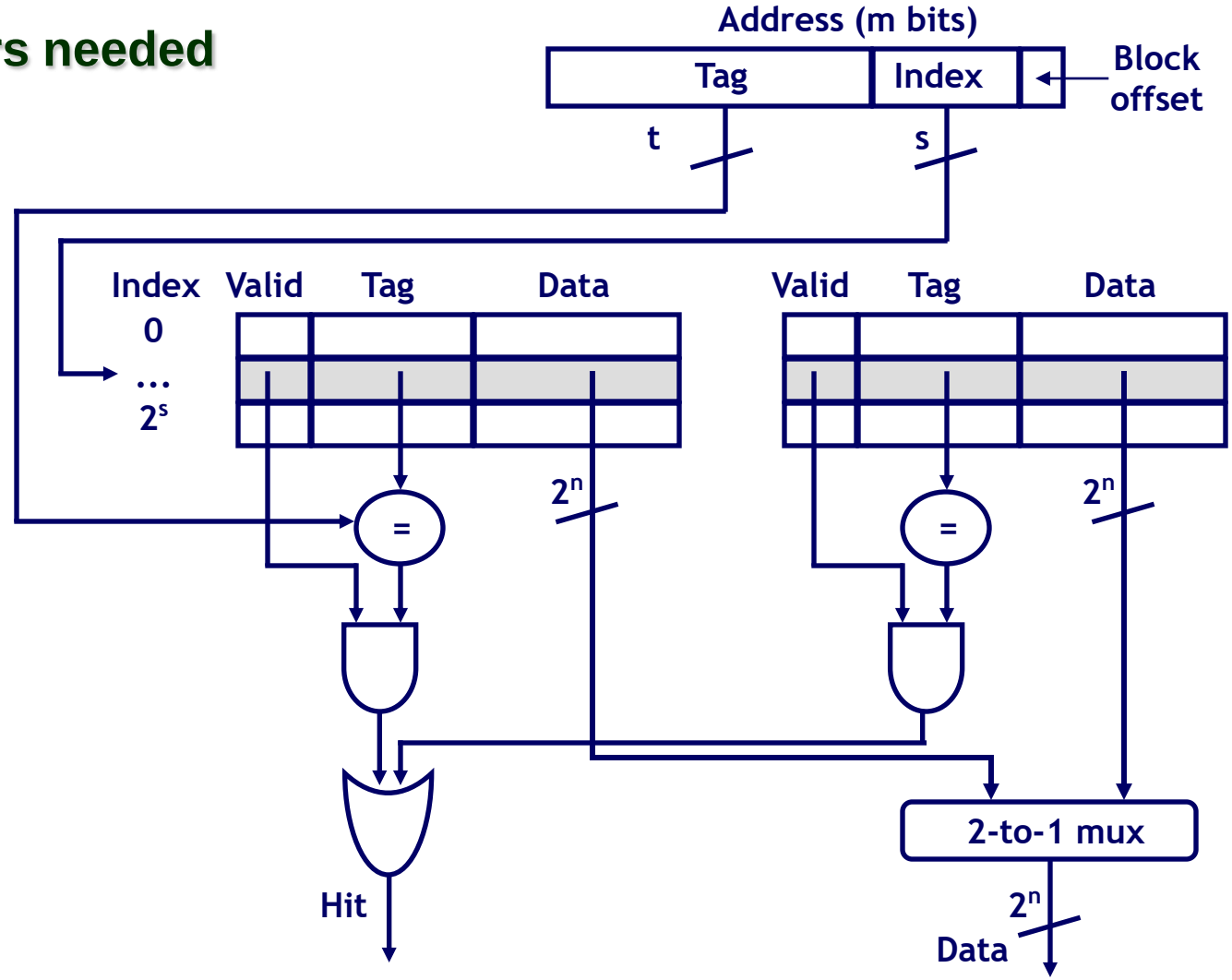
2-way associativity
4 sets, 2 blocks each



4-way associativity
2 sets, 4 blocks each

2-way set associative cache implementation

Two comparators needed



Practice problem 6.12

Consider a 2-way set associative cache (S,E,B,m) = (8,2,4,13)

Excluding the overhead of the tags and valid bits, what is the capacity of this cache?

64 bytes

Draw a figure of this cache

Set	Tag	Data0-3	Tag	Data0-3
0				
1				
2				
3				
4				
5				
6				
7				

Draw a diagram that shows the parts of the address that are used to determine the cache tag, cache set index, and cache block offset

Practice problem 6.13

Consider a 2-way set associative cache (S,E,B,m) = (8,2,4,13)

- Note: Invalid cache lines are left blank

Set	Tag	Data0-3	Tag	Data0-3
0	09	86 30 3F 10	00	
1	45	60 4F E0 23	38	00 BC 0B 37
2	EB		0B	
3	06		32	12 08 7B AD
4	C7	06 78 07 C5	05	40 67 C2 3B
5	71	0B DE 18 4B	6E	
6	91	A0 B7 26 2D	F0	
7	46		DE	12 C0 88 37

Consider an access to 0x0E34.

- What is the block offset of this address in hex? 00 = 0x0
- What is the set index of this address in hex? 101 = 0x5
- What is the cache tag of this address in hex? 01110001 = 0x71
- Does this access hit or miss in the cache? = Hit
- What value is returned if it is a hit? = 0x0B

t (8 bits)	s (3 bits)	b (2 bits)

Practice problem 6.14

Consider a 2-way set associative cache (S,E,B,m) = (8,2,4,13)

- Note: Invalid cache lines are left blank

Set	Tag	Data0-3	Tag	Data0-3
0	09	86 30 3F 10	00	
1	45	60 4F E0 23	38	00 BC 0B 37
2	EB		0B	
3	06		32	12 08 7B AD
4	C7	06 78 07 C5	05	40 67 C2 3B
5	71	0B DE 18 4B	6E	
6	91	A0 B7 26 2D	F0	
7	46		DE	12 C0 88 37

Consider an access to 0x0DD5.

- What is the block offset of this address in hex? 01 = 0x1
- What is the set index of this address in hex? 101 = 0x5
- What is the cache tag of this address in hex? 01101110 = 0x6E
- Does this access hit or miss in the cache? Miss (invalid block)
- What value is returned if it is a hit?

t (8 bits)	s (3 bits)	b (2 bits)

Practice problem 6.16

Consider a 2-way set associative cache (S,E,B,m) = (8,2,4,13)

- **Note: Invalid cache lines are left blank**

Set	Tag	Data0-3	Tag	Data0-3
0	09	86 30 3F 10	00	
1	45	60 4F E0 23	38	00 BC 0B 37
2	EB		0B	
3	06		32	12 08 7B AD
4	C7	06 78 07 C5	05	40 67 C2 3B
5	71	0B DE 18 4B	6E	
6	91	A0 B7 26 2D	F0	
7	46		DE	12 C0 88 37

List all hex memory addresses that will hit in Set 3

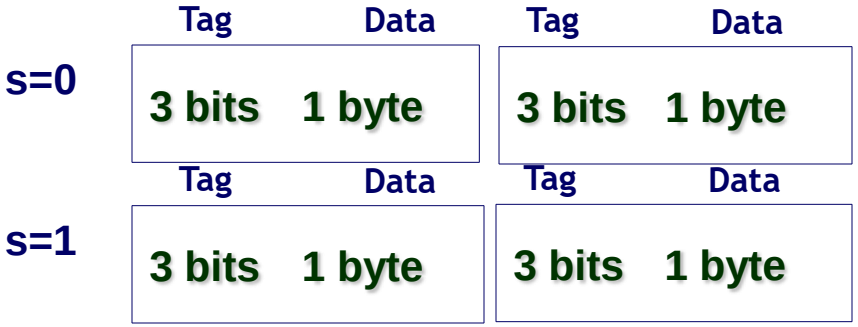
00110010011xx = 0x64C, 0x64D, 0x64E, 0x64F

Practice problem

Consider the following set associative cache: (S,E,B,m) = (2,2,1,4)

- Derive values for number of address bits used for the tag (t), the index (s) and the block offset (b)
  - s = 1
  - b = 0
  - t = 3
- Draw a diagram of which bits of the address are used for the tag, the set index and the block offset

- Draw a diagram of the cache



A fully associative cache

A fully associative cache permits data to be stored in *any* cache block, instead of forcing each memory address into one particular block.

- There is only 1 set (i.e. S=1 and s=0)
  - C = E*B
- When data is fetched from memory, it can be placed in *any* unused block of the cache.
- Eliminates conflict misses between two or more memory addresses which map to a single cache block.

Fully associative cache example

Consider the following fully associative cache: (S,E,B,m) = (1,4,1,4)

- Derive values for number of address bits used for the tag (t), the index (s) and the block offset (b)

  s = 0

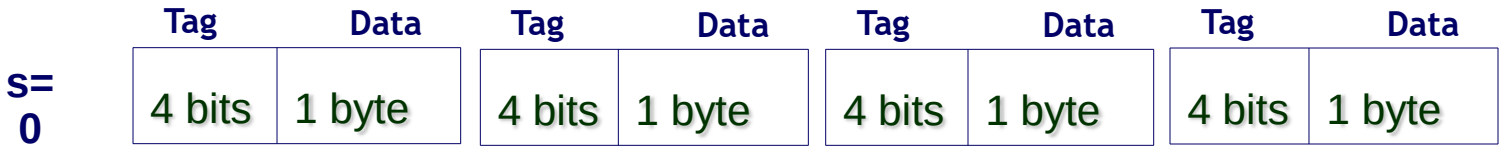
  b = 0

  t = 4

- Draw a diagram of which bits of the address are used for the tag, the set index and the block offset

  t (4 bits)

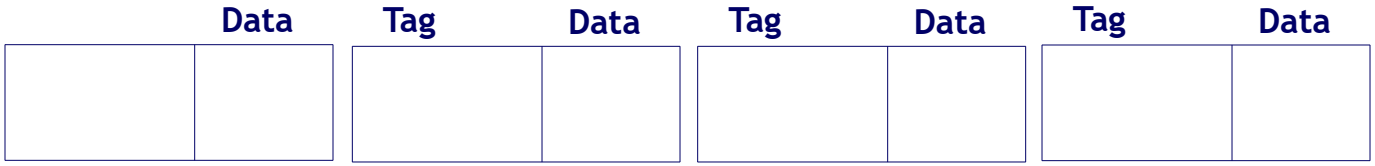
- Draw a diagram of the cache



Fully associative cache (S,E,B,m) = (1,4,1,4)

0000	0x00
0001	0x11
0010	0x22
0011	0x33
0100	0x44
0101	0x55
0110	0x66
0111	0x77
1000	0x88
1001	0x99
1010	0xAA
1011	0xBB
1100	0xCC
1101	0xDD
1110	0xEE
1111	0xFF

Cache



Access pattern:
0010
0110
0010
0110
0010
0110
...

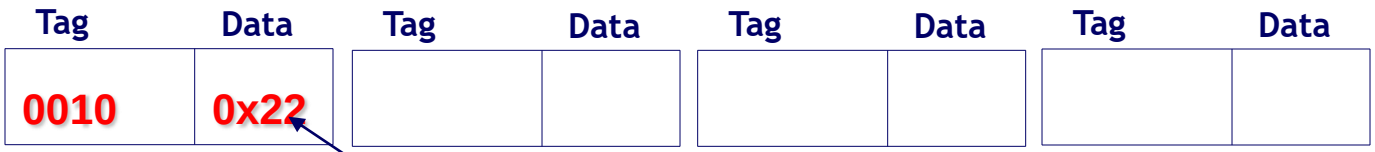
t (4 bits)

Fully associative cache (S,E,B,m) = (1,4,1,4)

Main memory

0000	0x00
0001	0x11
0010	0x22
0011	0x33
0100	0x44
0101	0x55
0110	0x66
0111	0x77
1000	0x88
1001	0x99
1010	0xAA
1011	0xBB
1100	0xCC
1101	0xDD
1110	0xEE
1111	0xFF

Cache



Access pattern:
0010
0110
0010
0110
0010
0110
...

Cache miss

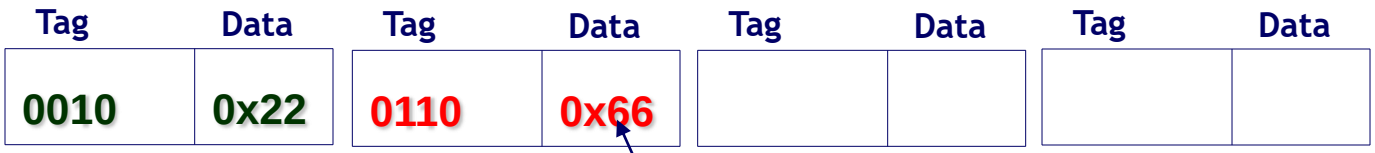
t (4 bits)

Fully associative cache (S,E,B,m) = (1,4,1,4)

Main memory

0000	0x00
0001	0x11
0010	0x22
0011	0x33
0100	0x44
0101	0x55
0110	0x66
0111	0x77
1000	0x88
1001	0x99
1010	0xAA
1011	0xBB
1100	0xCC
1101	0xDD
1110	0xEE
1111	0xFF

Cache



Access pattern:
0010
0110
0010
0110
0010
0110
…

Cache miss

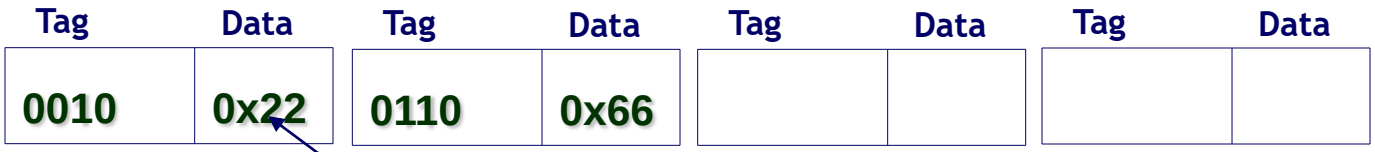
t (4 bits)

Fully associative cache (S,E,B,m) = (1,4,1,4)

Main memory

0000	0x00
0001	0x11
0010	0x22
0011	0x33
0100	0x44
0101	0x55
0110	0x66
0111	0x77
1000	0x88
1001	0x99
1010	0xAA
1011	0xBB
1100	0xCC
1101	0xDD
1110	0xEE
1111	0xFF

Cache



Access pattern:
0010
0110
0010
0110
0010
0110
...

Cache hit

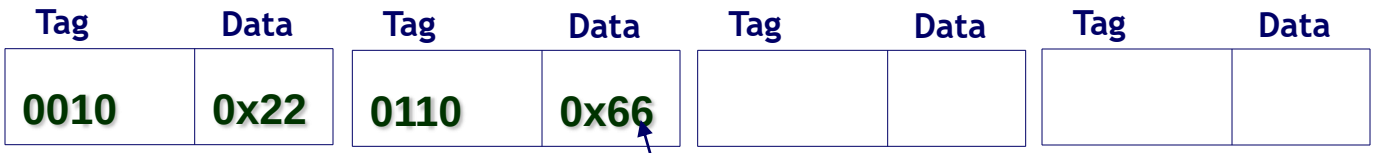
t (4 bits)

Fully associative cache (S,E,B,m) = (1,4,1,4)

Main memory

0000	0x00
0001	0x11
0010	0x22
0011	0x33
0100	0x44
0101	0x55
0110	0x66
0111	0x77
1000	0x88
1001	0x99
1010	0xAA
1011	0xBB
1100	0xCC
1101	0xDD
1110	0xEE
1111	0xFF

Cache



Access pattern:
0010
0110
0010
0110
0010
0110
...

Cache hit

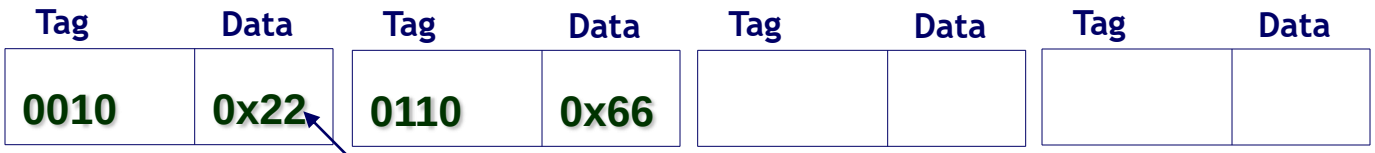
t (4 bits)

Fully associative cache (S,E,B,m) = (1,4,1,4)

Main memory

0000	0x00
0001	0x11
0010	0x22
0011	0x33
0100	0x44
0101	0x55
0110	0x66
0111	0x77
1000	0x88
1001	0x99
1010	0xAA
1011	0xBB
1100	0xCC
1101	0xDD
1110	0xEE
1111	0xFF

Cache



Access pattern:
0010
0110
0010
0110
0010
0110
...

Cache hit

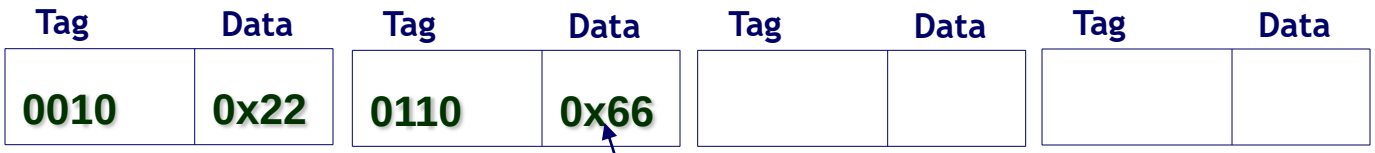
t (4 bits)

Fully associative cache (S,E,B,m) = (1,4,1,4)

Main memory

0000	0x00
0001	0x11
0010	0x22
0011	0x33
0100	0x44
0101	0x55
0110	0x66
0111	0x77
1000	0x88
1001	0x99
1010	0xAA
1011	0xBB
1100	0xCC
1101	0xDD
1110	0xEE
1111	0xFF

Cache



Access pattern:
0010
0110
0010
0110
0010
0110
…

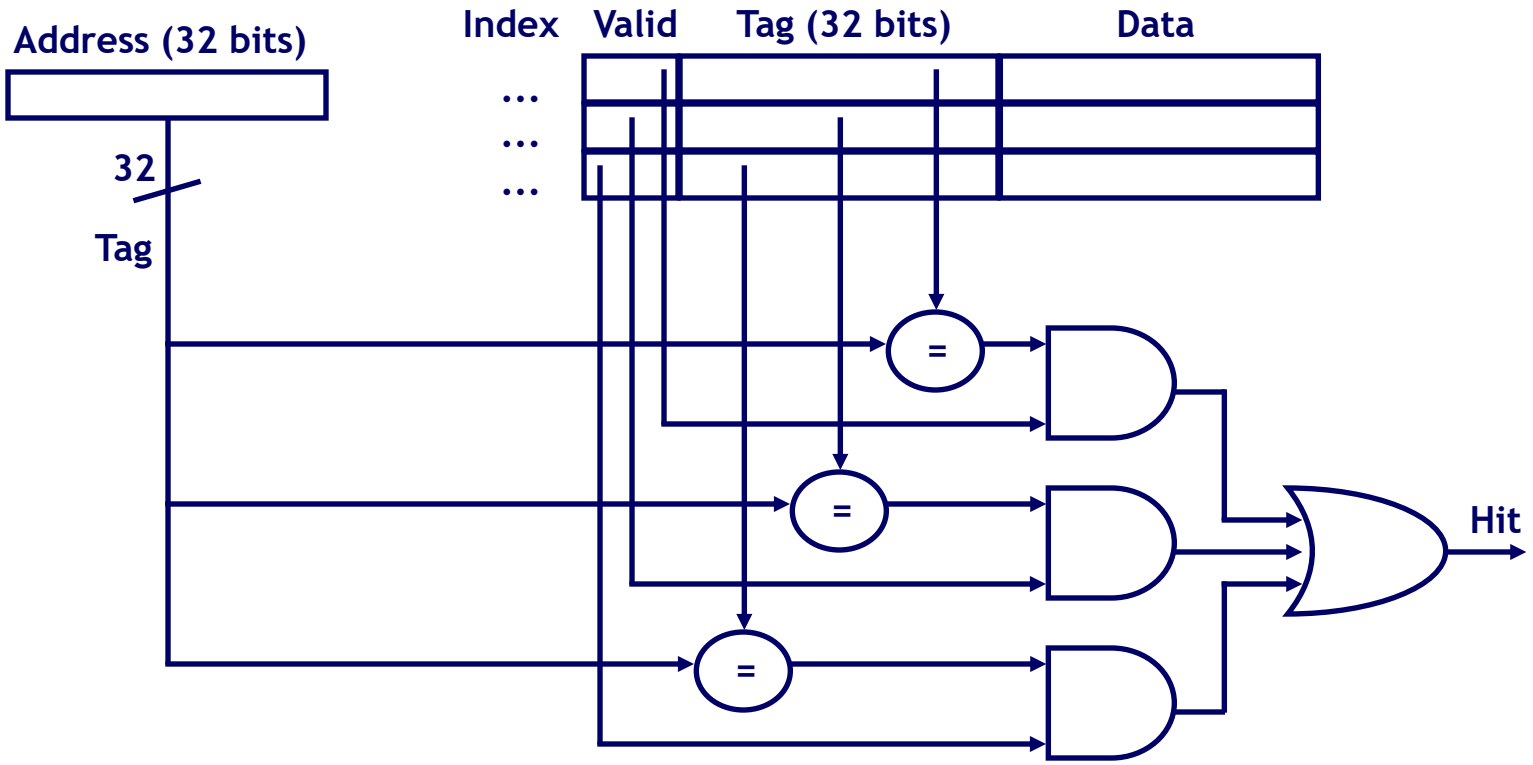
Cache hit

Miss rate after first two cold misses = 0%

The price of full associativity

However, a fully associative cache is expensive to implement.

- **No index field in the address anymore**
- ***Entire* address used as the tag, increasing the total cache size.**
- **Data could be anywhere in the cache, so we must check the tag of *every* cache block. That's a lot of comparators!**



Cache design

Size

Associativity

Block Size

Replacement Algorithm

Write Policy

Number of Caches

Block (cache line) size

8 to 64 bytes is typical

Advantages of large block size?

Disadvantages of large block size?

Replacement algorithms

Any empty block in the correct set may be used for storing data.

If there are no empty blocks, the cache will attempt to replace one

Most common LRU (least recently used)

- Assuming a block hasn't been used in a while, it won't be needed again anytime soon.

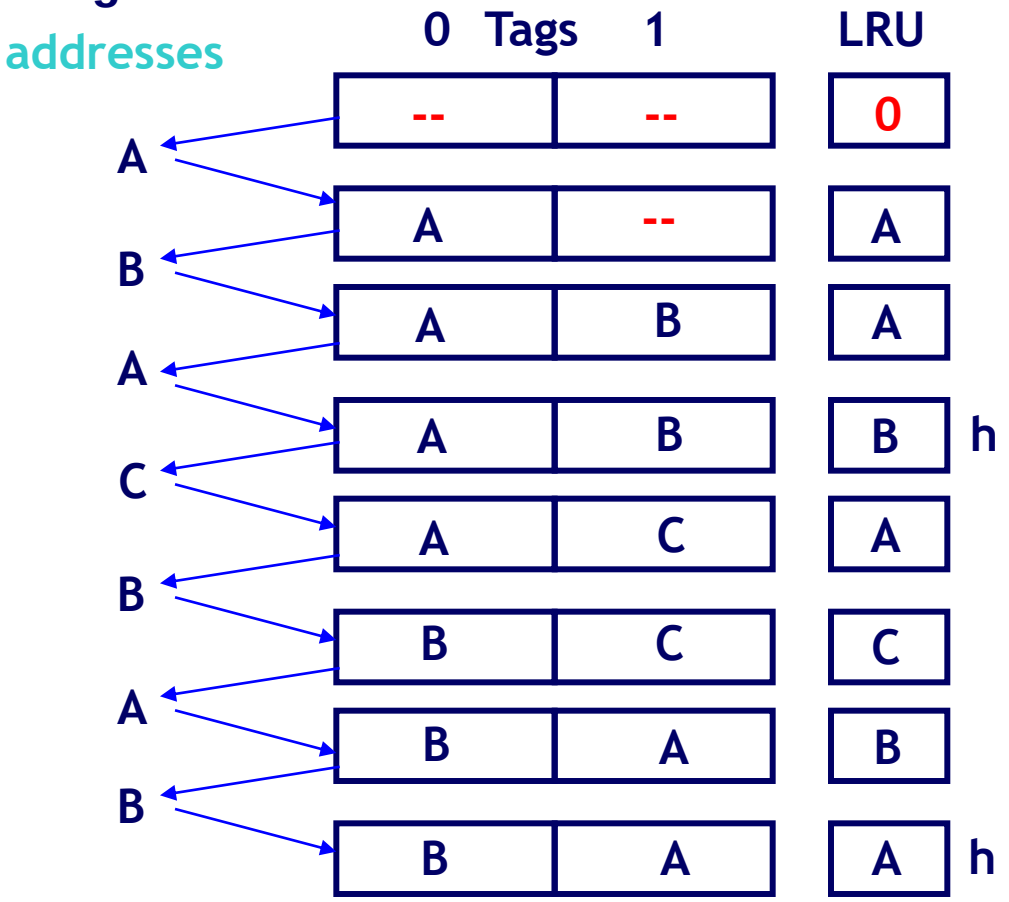
Others

- First in first out (FIFO)
  - replace block that has been in cache longest
- Least frequently used
  - replace block which has had fewest hits
- Random

LRU example

Assume a fully-associative cache with two blocks, which of the following memory references miss in the cache.

- assume distinct addresses go to distinct blocks

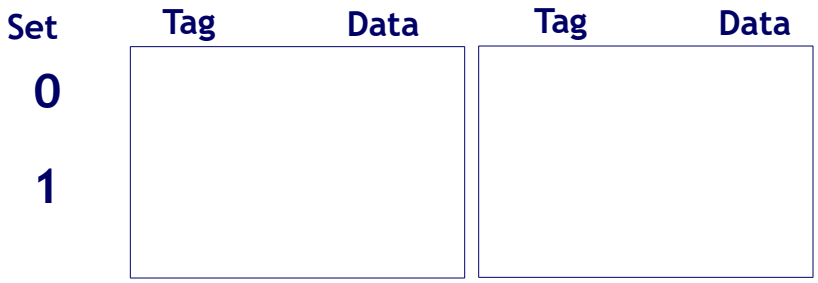


2-way set associative LRU cache (S,E,B,m) = (2,2,1,4)

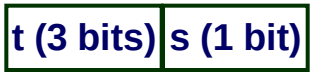
Main memory

0000	0x00
0001	0x11
0010	0x22
0011	0x33
0100	0x44
0101	0x55
0110	0x66
0111	0x77
1000	0x88
1001	0x99
1010	0xAA
1011	0xBB
1100	0xCC
1101	0xDD
1110	0xEE
1111	0xFF

Cache



Access pattern:
0010
0110
0010
0000
0010
0110
...

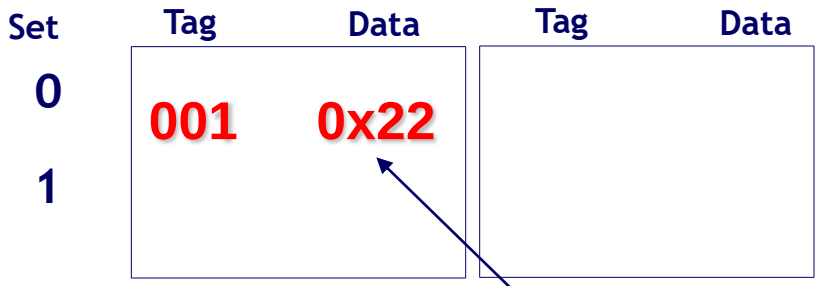


2-way set associative LRU cache (S,E,B,m) = (2,2,1,4)

Main memory

0000	0x00
0001	0x11
0010	0x22
0011	0x33
0100	0x44
0101	0x55
0110	0x66
0111	0x77
1000	0x88
1001	0x99
1010	0xAA
1011	0xBB
1100	0xCC
1101	0xDD
1110	0xEE
1111	0xFF

Cache



Access pattern:
0010
0110
0010
0000
0010
0110
...

Cache miss

t (3 bits) | s (1 bit)

2-way set associative LRU cache (S,E,B,m) = (2,2,1,4)

Main memory

0000	0x00
0001	0x11
0010	0x22
0011	0x33
0100	0x44
0101	0x55
0110	0x66
0111	0x77
1000	0x88
1001	0x99
1010	0xAA
1011	0xBB
1100	0xCC
1101	0xDD
1110	0xEE
1111	0xFF

Cache

Set	Tag	Data	Tag	Data
0	001	0x22	011	0x66
1				

Access pattern:
0010
0110
0010
0000
0010
0110
...

Cache miss

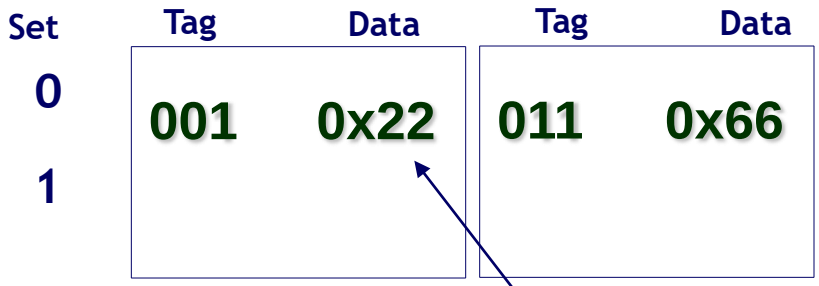
t (3 bits)	s (1 bit)

2-way set associative LRU cache (S,E,B,m) = (2,2,1,4)

Main memory

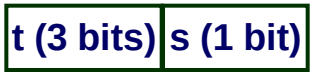
0000	0x00
0001	0x11
0010	0x22
0011	0x33
0100	0x44
0101	0x55
0110	0x66
0111	0x77
1000	0x88
1001	0x99
1010	0xAA
1011	0xBB
1100	0xCC
1101	0xDD
1110	0xEE
1111	0xFF

Cache



Access pattern:
0010
0110
0010
0000
0010
0110
...

Cache hit



2-way set associative LRU cache (S,E,B,m) = (2,2,1,4)

Main memory

0000	0x00
0001	0x11
0010	0x22
0011	0x33
0100	0x44
0101	0x55
0110	0x66
0111	0x77
1000	0x88
1001	0x99
1010	0xAA
1011	0xBB
1100	0xCC
1101	0xDD
1110	0xEE
1111	0xFF

Cache

Set	Tag	Data	Tag	Data
0	001	0x22	000	0x00
1				

Access pattern:
0010
0110
0010
0000
0010
0110
...

Cache miss (replace LRU)

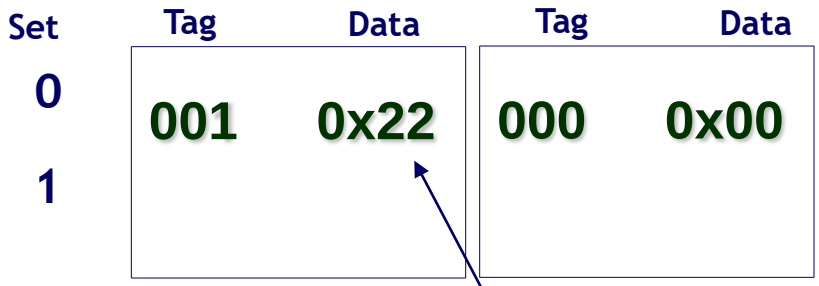
t (3 bits)	s (1 bit)

2-way set associative LRU cache (S,E,B,m) = (2,2,1,4)

Main memory

0000	0x00
0001	0x11
0010	0x22
0011	0x33
0100	0x44
0101	0x55
0110	0x66
0111	0x77
1000	0x88
1001	0x99
1010	0xAA
1011	0xBB
1100	0xCC
1101	0xDD
1110	0xEE
1111	0xFF

Cache



Access pattern:
0010
0110
0010
0000
0010
0110
...

Cache hit

t (3 bits) | s (1 bit)

2-way set associative LRU cache (S,E,B,m) = (2,2,1,4)

Main memory

0000	0x00
0001	0x11
0010	0x22
0011	0x33
0100	0x44
0101	0x55
0110	0x66
0111	0x77
1000	0x88
1001	0x99
1010	0xAA
1011	0xBB
1100	0xCC
1101	0xDD
1110	0xEE
1111	0xFF

Cache

Set	Tag	Data	Tag	Data
0	001	0x22	011	0x66
1				

Access pattern:
0010
0110
0010
0000
0010
0110
...

Cache miss

t (3 bits)	s (1 bit)

Write-policies

What happens when data is written to memory through cache

Two cases

- Write-hit: Block being written is in cache
- Write-miss: Block being written is not in cach

Write-hit policies

Write-through

- Writes go immediately to main memory (as well as cache)
- Multiple cores/CPUs can monitor main memory traffic to keep caches consistent and up to date
- Lots of traffic
- Slows down writes

Write-back

- Writes initially made in cache only
- Defer write to memory until replacement of line
  - Use a dirty bit to indicate cache entry that has been updated
  - If block is to be replaced, write to main memory only if dirty bit is set
- Other caches can get out of sync
- Some CPUs allow one to set the cache policy for a particular program or memory page

Write miss policies

Write-allocate

- When write to address misses, load into cache
- Update line in cache
- Good if there are subsequent reads and writes to address

No write-allocate

- When write to address misses, bypass cache and write straight to memory
- Do not load into cache
- Good if there are no subsequent reads and writes to address

Typically,

- Write-through policy paired with no-write-allocate policy
- Write-back policy paired with write-allocate policy

Number of caches

As logic density increases, it has become advantages and practical to create multi-level caches:

- **on chip**
- **off chip**

Examples of a deepening hierarchy

- **Pentium II: L1 (on chip) & L2 (off chip)**
- **Pentium III: L1 and L2 (on-chip), L3 (off chip)**
- **Intel Core: L1 and L2 per core (on-chip), Unified L3 (on-chip)**

Number of caches

Another way to add caches is to split them based on what they store

- Data cache
- Program cache

Advantage:

- Likely increased hit rates – data and program accesses display different behavior

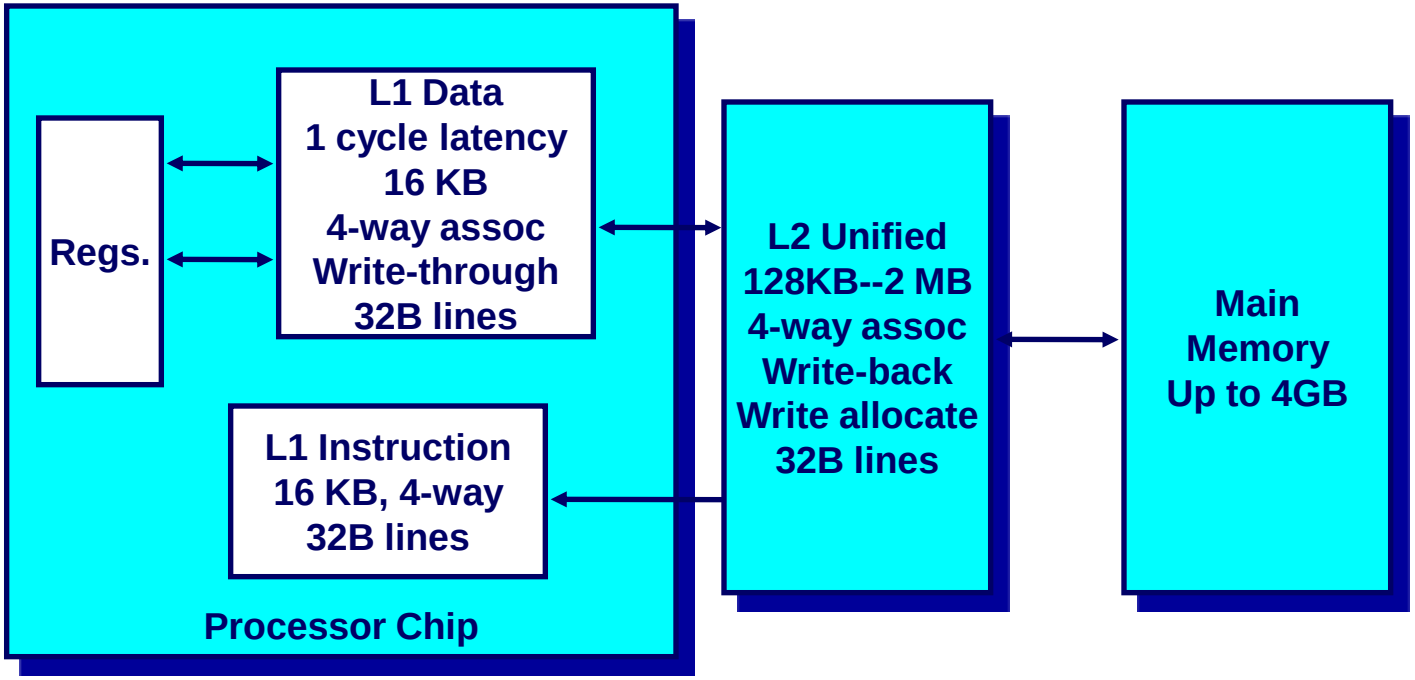
Disadvantage

- Complexity

Split cache example

Separate data and instruction caches vs. unified cache

Intel Pentium Cache Hierarchy



Querying cache parameters

Exposed via CPUID x86 instruction

- getconf –a | egrep LEVEL in Linux
- grep . /sys/devices/system/cpu/cpu0/cache/index*/*

Metric	Nehalem	Sandy Bridge	Haswell
L1 Instruction Cache	32K, 4-way	32K, 8-way	32K, 8-way
L1 Data Cache	32K, 8-way	32K, 8-way	32K, 8-way
Fastest Load-to-use	4 cycles	4 cycles	4 cycles
Load bandwidth	16 Bytes/cycle	32 Bytes/cycle (banked)	64 Bytes/cycle
Store bandwidth	16 Bytes/cycle	16 Bytes/cycle	32 Bytes/cycle
L2 Unified Cache	256K, 8-way	256K, 8-way	256K, 8-way
Fastest load-to-use	10 cycles	11 cycles	11 cycles
Bandwidth to L1	32 Bytes/cycle	32 Bytes/cycle	64 Bytes/cycle
L1 Instruction TLB	4K: 128, 4-way 2M/4M: 7/thread	4K: 128, 4-way 2M/4M: 8/thread	4K: 128, 4-way 2M/4M: 8/thread
L1 Data TLB	4K: 64, 4-way 2M/4M: 32, 4-way 1G: fractured	4K: 64, 4-way 2M/4M: 32, 4-way 1G: 4, 4-way	4K: 64, 4-way 2M/4M: 32, 4-way 1G: 4, 4-way
L2 Unified TLB	4K: 512, 4-way	4K: 512, 4-way	4K+2M shared: 1024, 8-way

All caches use 64-byte lines

Memory Mountain

Measuring memory performance of a system

Metric used: Read throughput (read bandwidth)

- Number of bytes read from memory per second (MB/s)
- Measured read throughput is a function of spatial and temporal locality.
- Compact way to characterize memory system performance

Two parameters

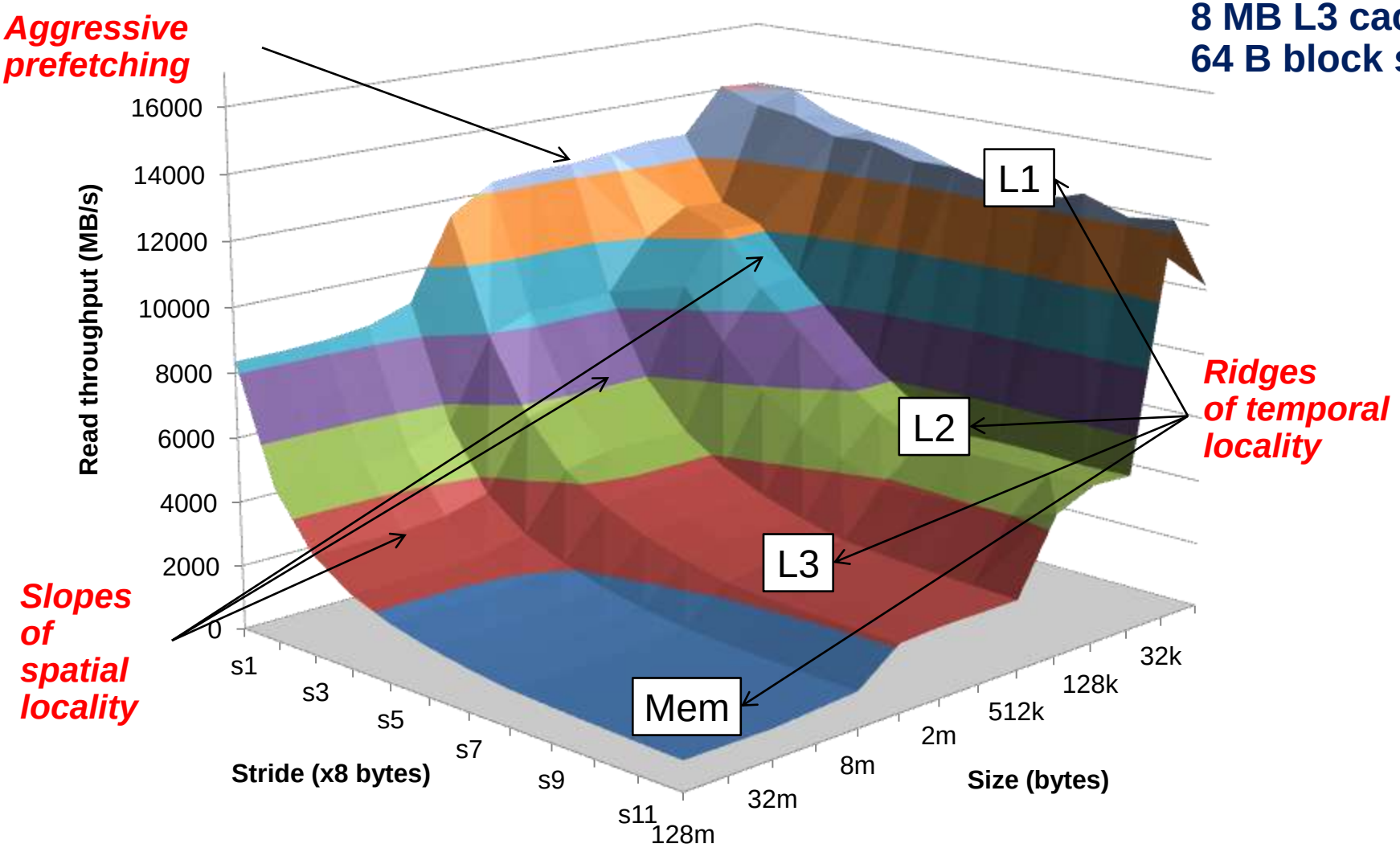
- Size of data set (`elems`)
- Access stride length (`stride`)

Memory mountain test function

```
long data[MAXELEMS];  /* Global array to traverse */

/* test - Iterate over first "elems" elements of
 *        array "data" with stride of "stride", using
 *        using 4x4 loop unrolling.
 */
int test(int elems, int stride) {
    long i, sx2=stride*2, sx3=stride*3, sx4=stride*4;
    long acc0 = 0, acc1 = 0, acc2 = 0, acc3 = 0;
    long length = elems, limit = length - sx4;

    /* Combine 4 elements at a time */
    for (i = 0; i < limit; i += sx4) {
        acc0 = acc0 + data[i];
        acc1 = acc1 + data[i+stride];
        acc2 = acc2 + data[i+sx2];
        acc3 = acc3 + data[i+sx3];
    }

    /* Finish any remaining elements */
    for (; i < length; i++) {
        acc0 = acc0 + data[i];
    }
    return ((acc0 + acc1) + (acc2 + acc3));
}
```

Memory Mountain results

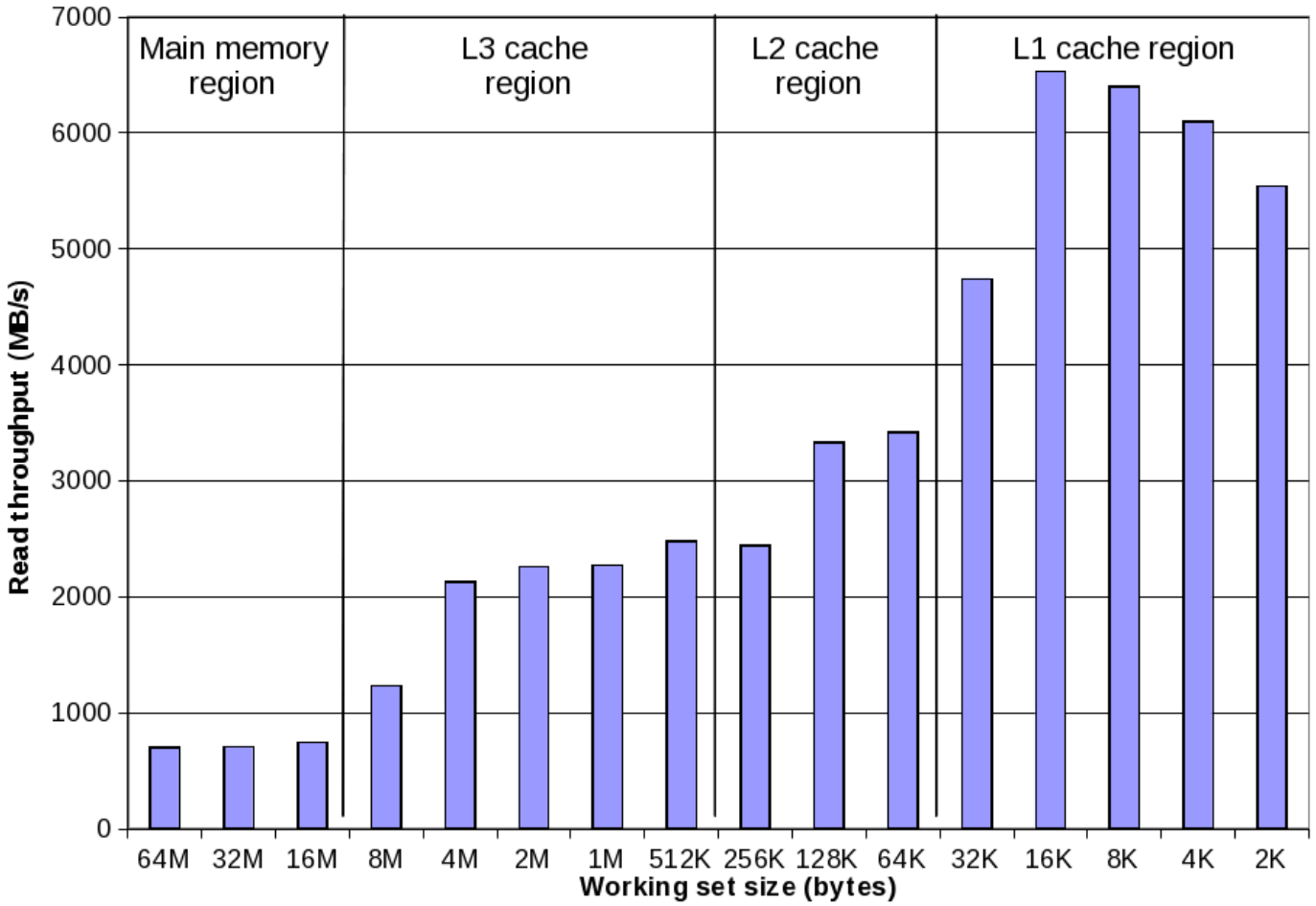
Core i7 Haswell
2.1 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size



Ridges of Temporal Locality

Slice through the memory mountain with stride=16

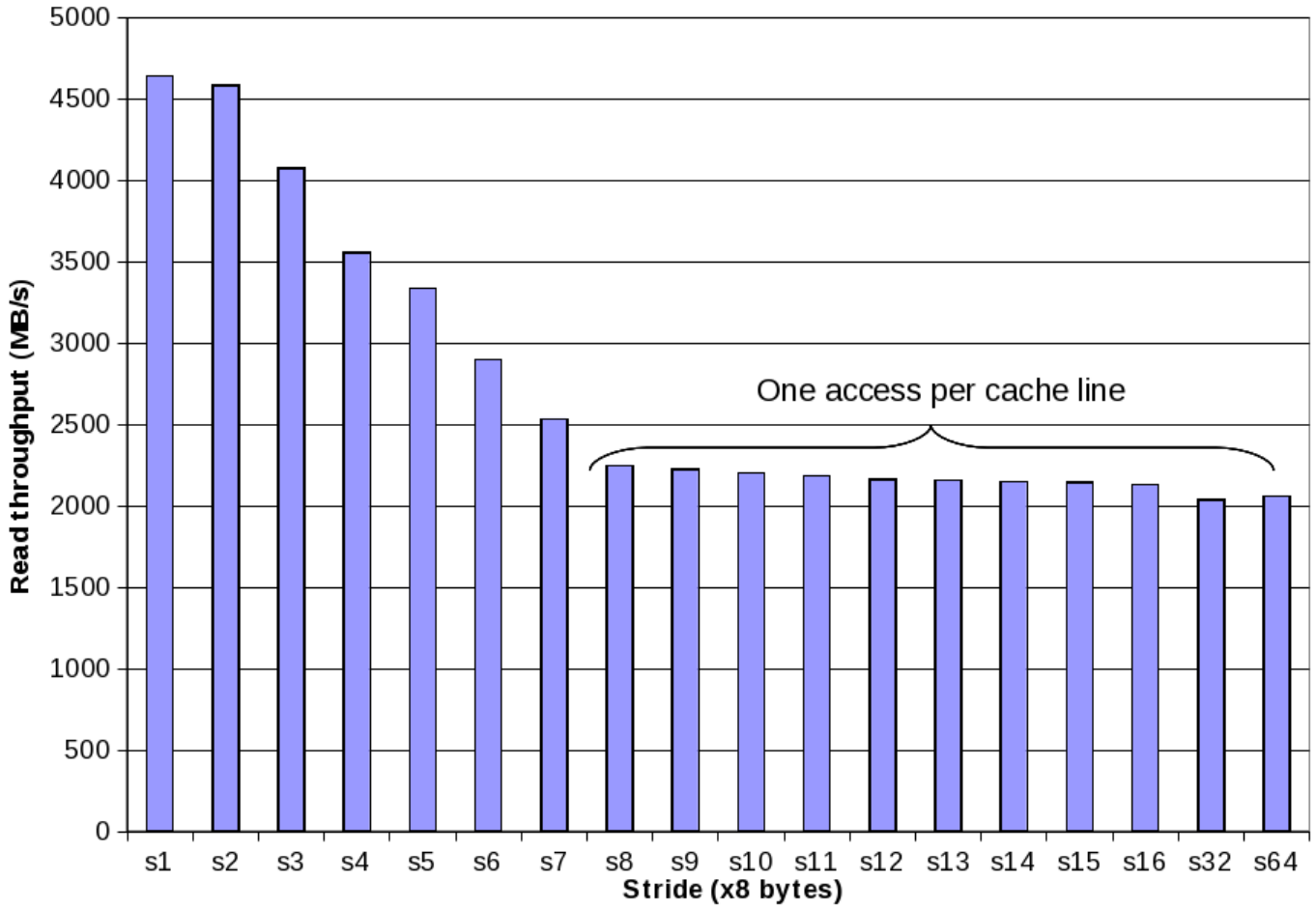
- Read throughputs of different caches and memory



A Slope of Spatial Locality

Slice through memory mountain with size=4MB

- Cache block size.



Matrix Multiplication Example

Writing memory efficient code

Cache effects to consider

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

```
/* ijk */                          Variable sum
for (i=0; i<n; i++)  {             held in register
  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;
  }
}
```

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 register

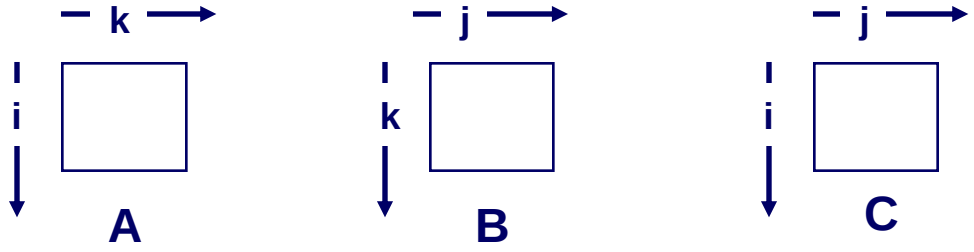
Miss Rate Analysis for Matrix Multiply

Assume:

- Double-precision floating point numbers
- Line size = 32 bytes (big enough for 4 8-byte doubles)
- 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



Layout of C Arrays in Memory (review)

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

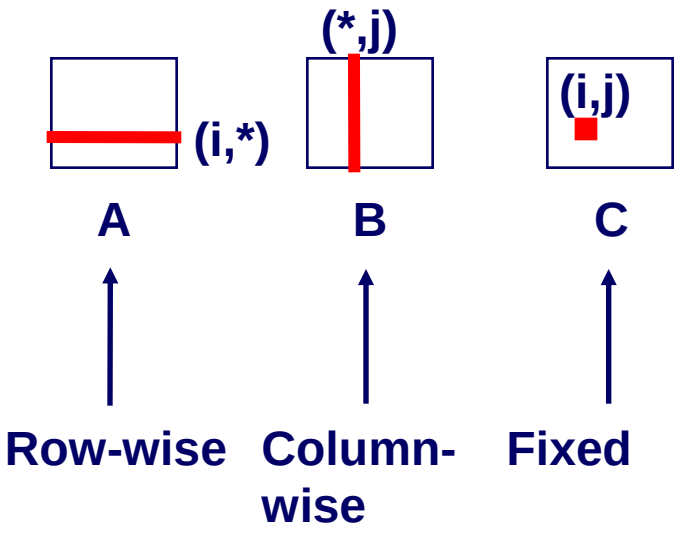
Stepping through rows in one column:

- `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 (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:

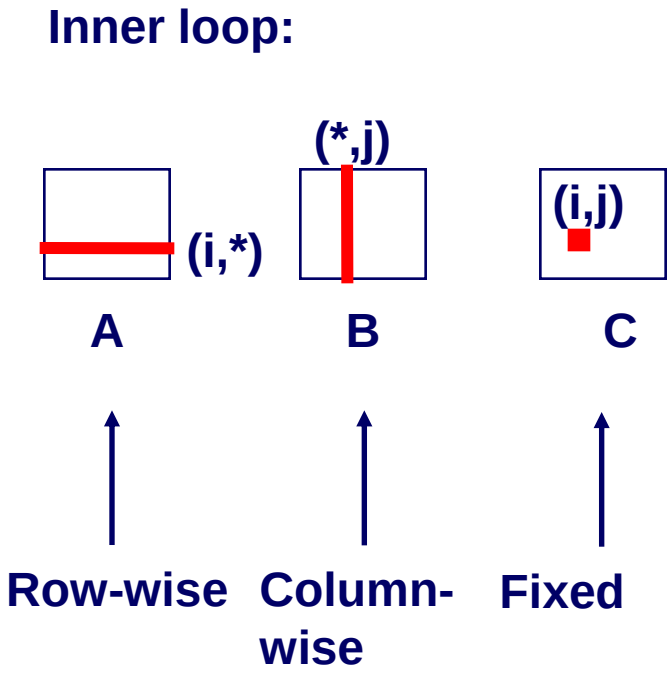


Misses per Inner Loop Iteration:

A	B	C
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
  }
}
```

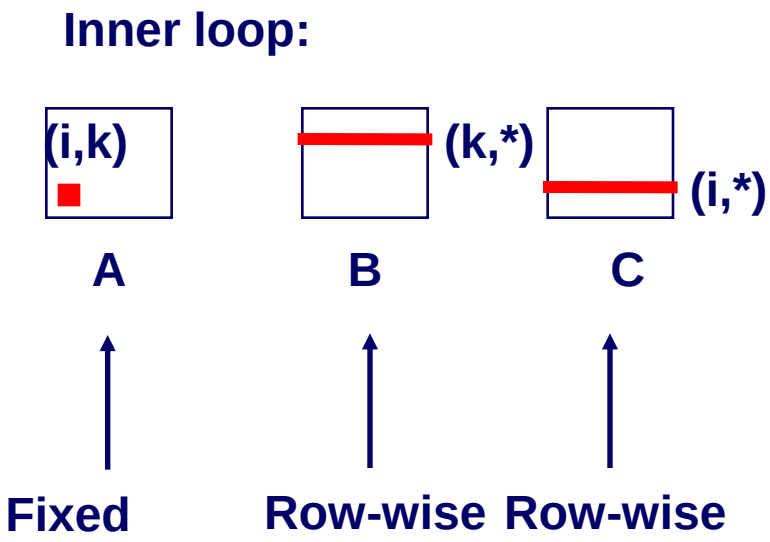


Misses per Inner Loop Iteration:

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];
  }
}
```



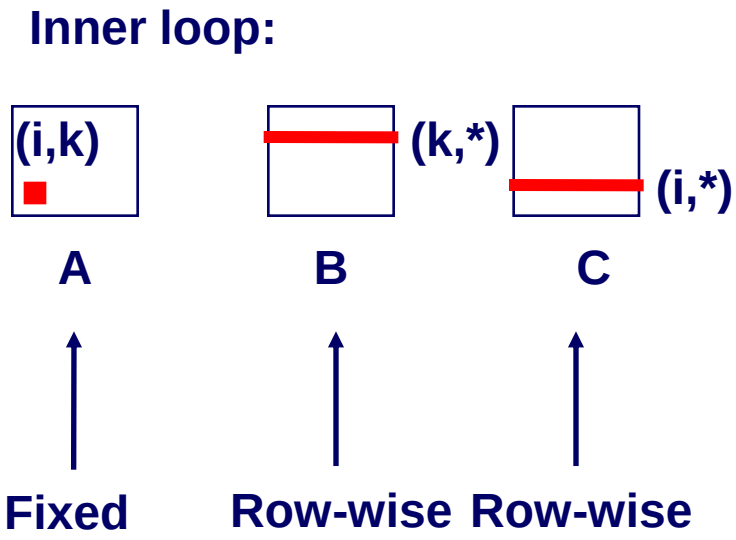
Misses per Inner Loop Iteration:

A	B	C
0.0	0.25	0.25

Better read performance, but now incur a memory store per iteration

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];
  }
}
```



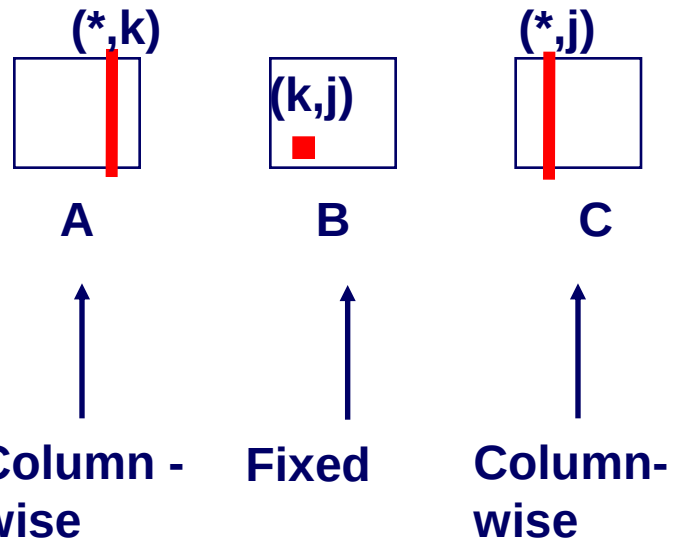
Misses per Inner Loop Iteration:

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:

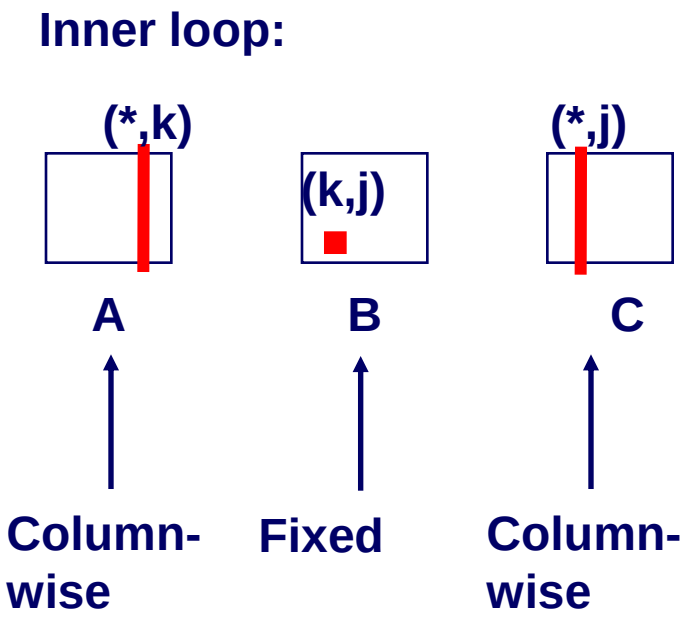


Misses per Inner Loop Iteration:

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

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

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

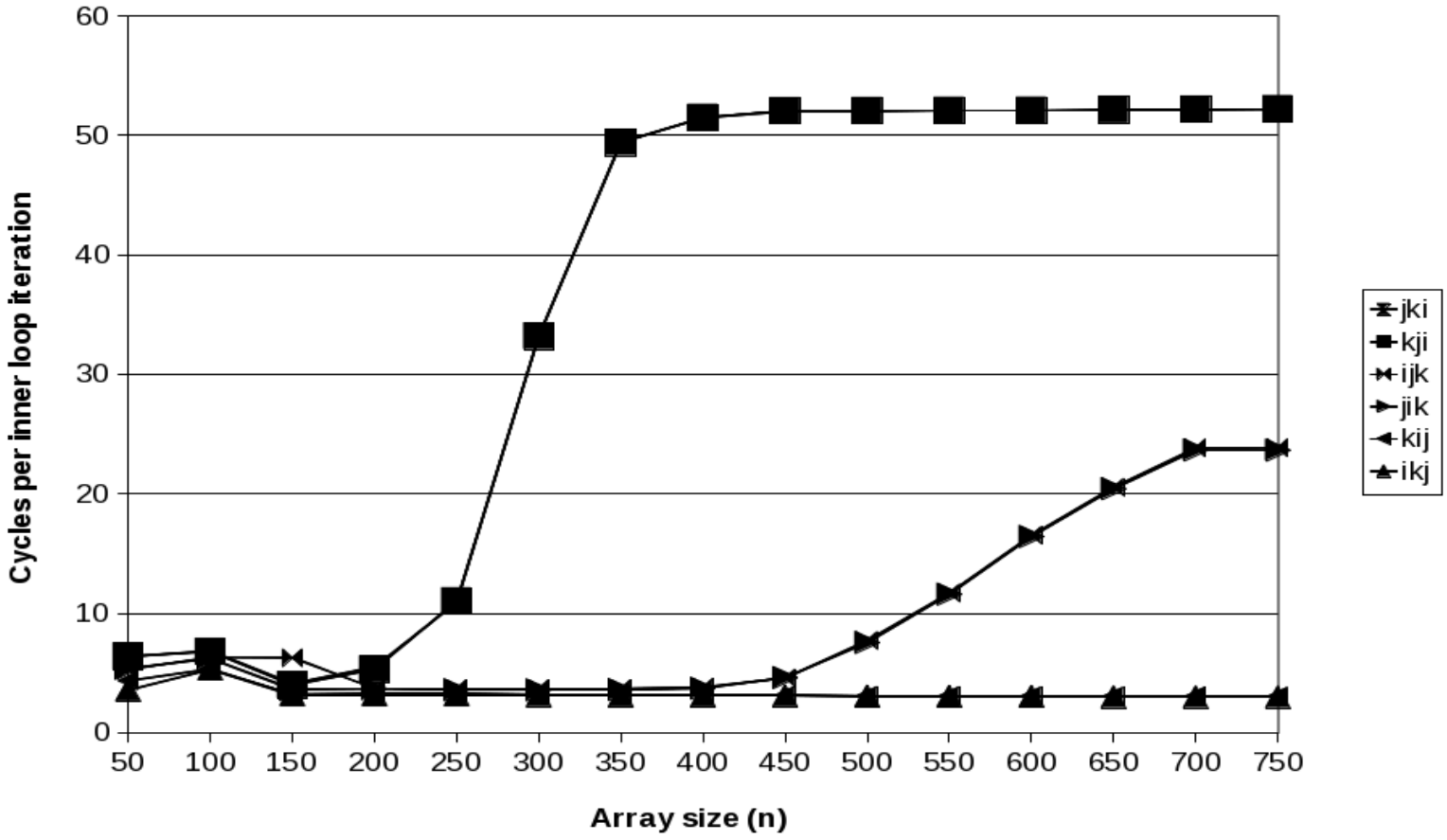
jki (& kji):
- 2 loads, *1 store*
- misses/iter = 2.0

```
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;
    }
}
```

```
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];
    }
}
```

```
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;
    }
}
```

Intel Core i7 Matrix Multiply Performance



Matrix multiplication performance

- Take maximal advantage of spatial locality in computation
- Size of matrices makes leveraging temporal locality difficult

How can one address this?

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

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

<u>Key idea:</u> 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)

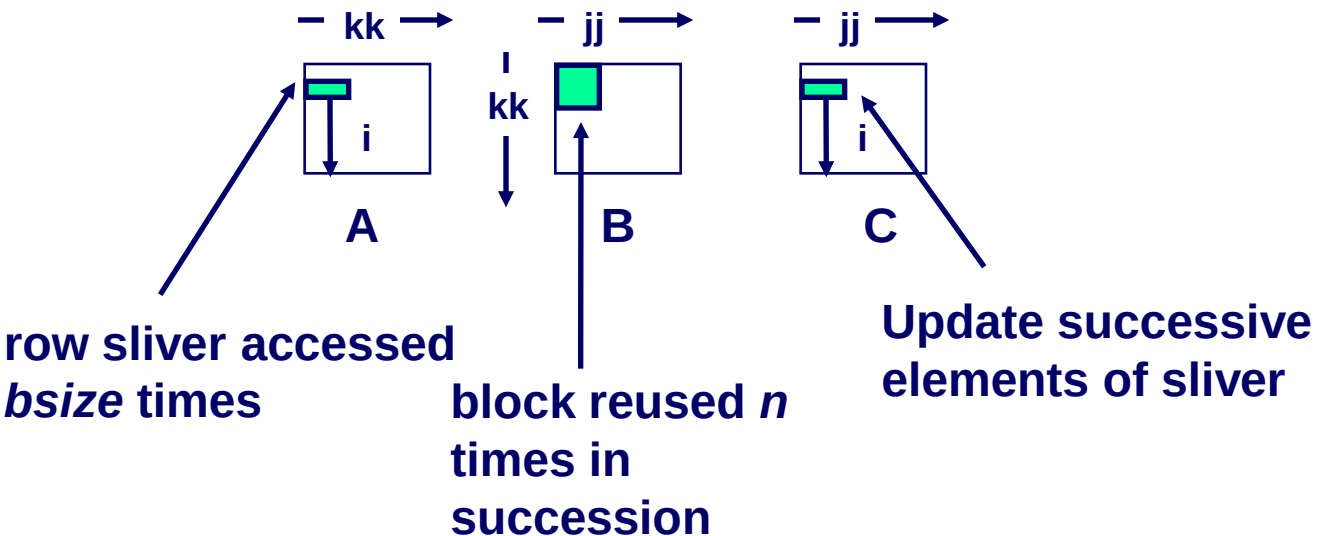
```
for (jj=0; jj<n; jj+=bsize) {
  for (i=0; i<n; i++)
    for (j=jj; j < min(jj+bsize,n); j++)
      c[i][j] = 0.0;
  for (kk=0; kk<n; kk+=bsize) {
    for (i=0; i<n; i++) {
      for (j=jj; j < min(jj+bsize,n); j++) {
        sum = 0.0
        for (k=kk; k < min(kk+bsize,n); k++) {
          sum +=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

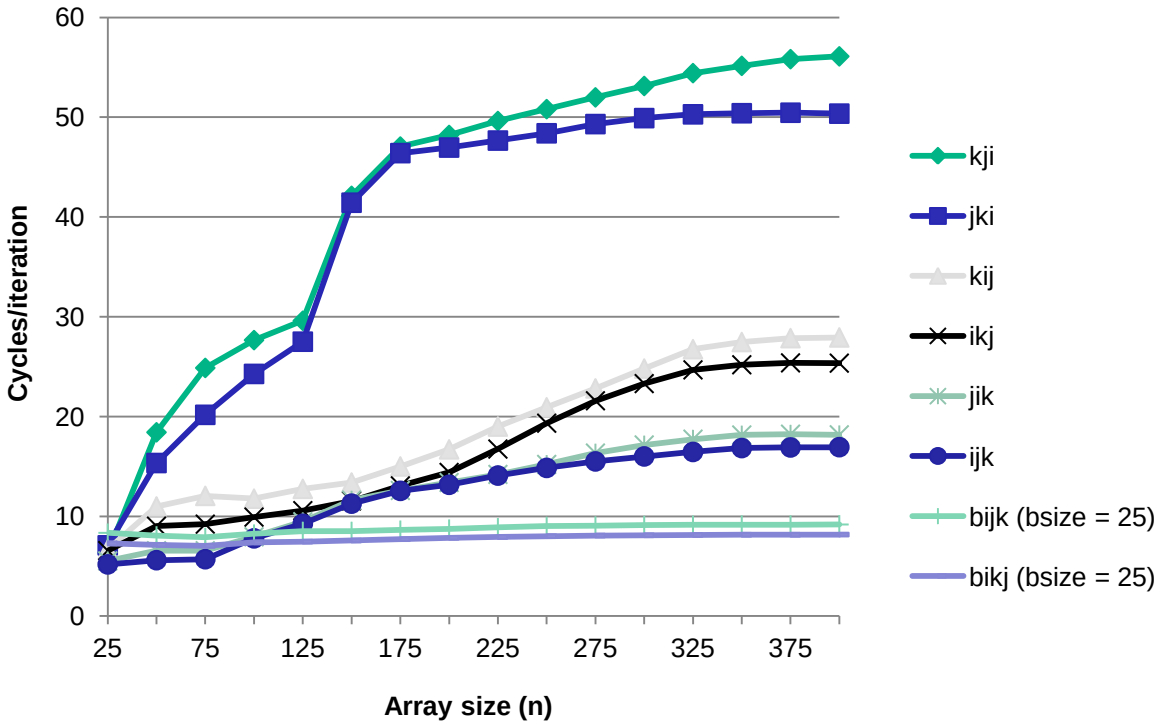
Innermost Loop Pair



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)

Exam practice

Chapter 6 Problems

6.7, 6.8	Stride-1 access in C
6.9	Cache parameters
6.10	Cache performance
6.11	Cache capacity
6.12, 6.13, 6.14, 6.15	Cache lookup
6.16	Cache address mapping
6.18	Cache simulation

Extra

Practice problem 6.15

Consider a 2-way set associative cache (S,E,B,m) = (8,2,4,13)

- Note: Invalid cache lines are left blank

Set	Tag	Data0-3	Tag	Data0-3
0	09	86 30 3F 10	00	
1	45	60 4F E0 23	38	00 BC 0B 37
2	EB		0B	
3	06		32	12 08 7B AD
4	C7	06 78 07 C5	05	40 67 C2 3B
5	71	0B DE 18 4B	6E	
6	91	A0 B7 26 2D	F0	
7	46		DE	12 C0 88 37

Consider an access to 0x1FE4.

- What is the block offset of this address in hex? 0x0
- What is the set index of this address in hex? 0x1
- What is the cache tag of this address in hex? 0xFF
- Does this access hit or miss in the cache? Miss
- What value is returned if it is a hit?

t (8 bits)	s (3 bits)	b (2 bits)

Writing Cache Friendly Code

Repeated references to variables are good (temporal locality)

Stride-1 reference patterns are good (spatial locality)

Revisiting example

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

```
int sumarrayrows(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 sumarraycols(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%

Typical Cache Performance

Miss Rate

- Fraction of memory references not found in cache (misses/references) (1 – HitRate)
- Typical numbers:
  - 3-10% for L1
  - can be quite small (e.g., < 1%) 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:
  - 4 clock cycle for L1
  - 10 clock cycles for L2

Miss Penalty

- Additional time required because of a miss
  - Typically 50-200 cycles for main memory

Let's 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"