Chapter 6: Caching II

Chapter 6 Topics:

- Cache-friendly Code
- Types of caches:
  - Direct-mapped,
  - Set Associative,
  - Fully Associative

Announcements

- Recitation Ex #4 due Monday in recitation
  - Will release solutions Monday at 6 pm
- Midterm #2 is Tuesday Nov 11
  - Similar format as before, closed book, no electronics, …
  - Can bring one page front/back summary sheet, and the 5-page Midterm #1 Table Packet
  - Covers Section 2.4 (Floating Point), Section 3.12 (buffer/ stack overflow), Chapter 4 (4.1-4.5.8, but skip 4.2 and 4.3.4), and Chapter 5
  - Will release some practice problems – TAs will review in recitation.
  - TAs also have office hours Thursday & Friday
- Performance Lab on due Monday Nov 17
- Read Chapter 6 all sections

Recap…

Introduced caching

- Place a small amount of faster but costlier memory nearer the CPU, and cache commonly accessed data and instructions in it
- Speeds up overall performance provided a program exhibits locality
  - temporal locality of data & instructions
  - spatial locality of data & instructions
    - 1) arrays stored in row major order
    - 2) cache stores blocks of array elements
    - 3) access these elements sequentially, i.e. Stride-1

L1/L2/L3 Caching hierarchy

Cache hits and misses

Recap…

Stride-1 Access Pattern

```
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 = 25%

Stride-N Access Pattern

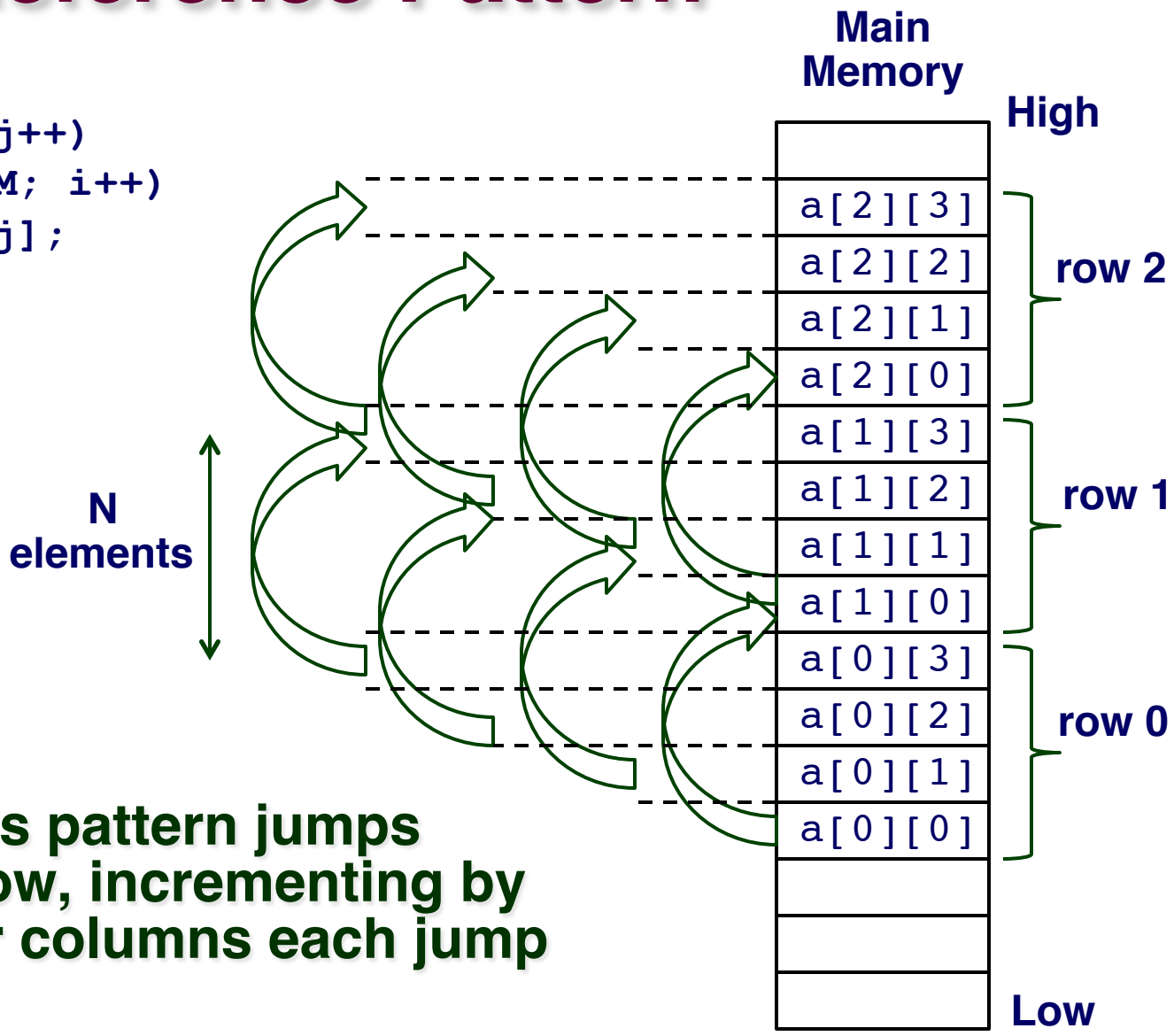
```
int sumarraycols(int a[M][N])
{
    int i, j, sum = 0;

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

Miss rate = 100%

Stride-N Reference Pattern

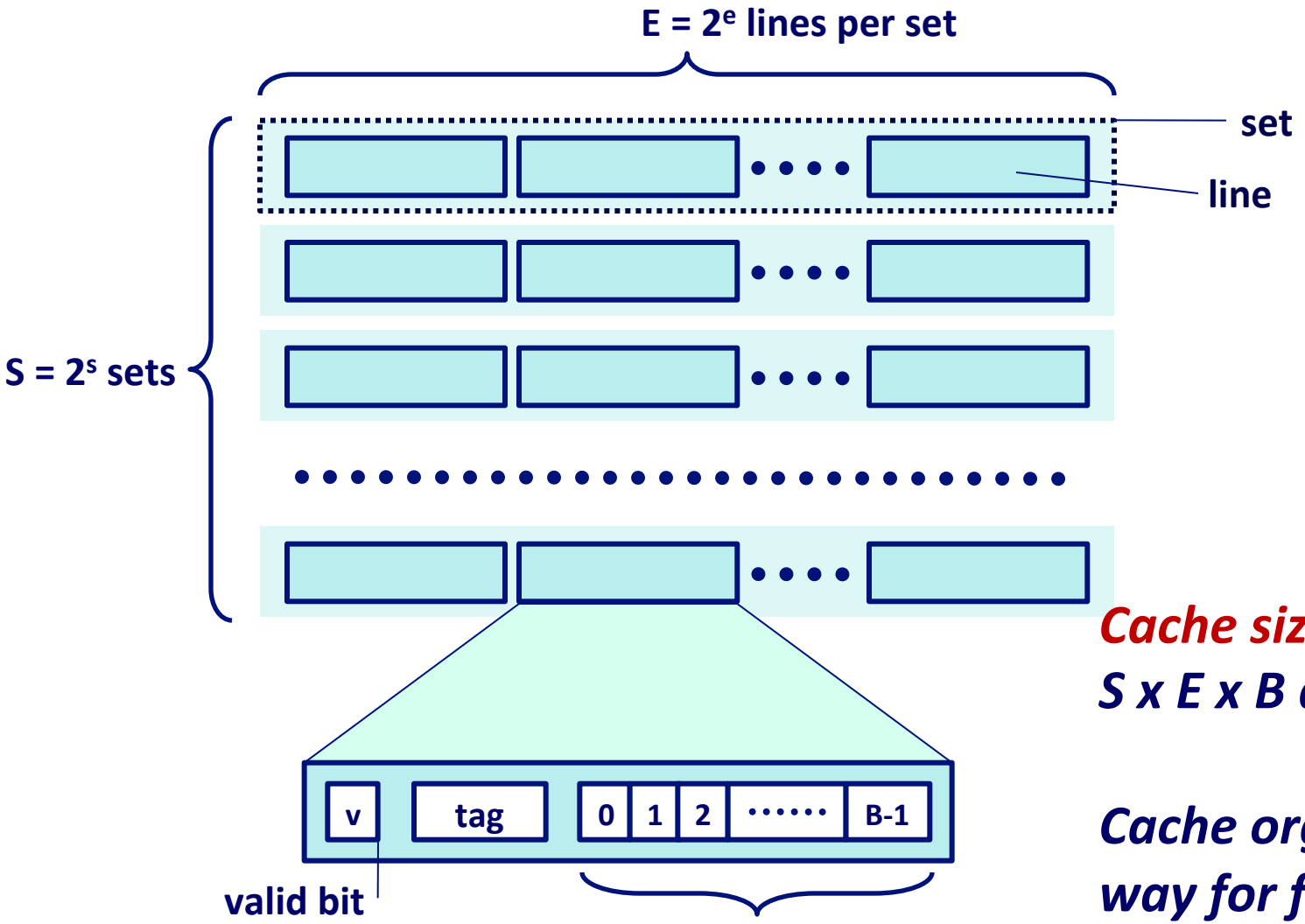
```
for (j = 0; j < N; j++)
    for (i = 0; i < M; i++)
        sum += a[i][j];
```



a Stride-N access pattern jumps from row to row, incrementing by N elements or columns each jump

General Cache Organization (S, E, B)

For L1 hardware caches, access must be fast, so organize as follows:



Cache size:
S x E x B data bytes

Cache organized this way for fast access

Stride-1, Caching & Spatial Locality

Stride-1:

- `for (i = 0; i < N; i++)`
  `sum += a[0][i];`
- accesses successive elements in memory
- 1st access to a[0][0] pulls in a cache block B worth of bytes containing a[0][0]
- If cache block size B > size of each word W in array a, then exploit spatial locality
  - i.e. cache block B contains not just a[0][0], but a[0][1], a[0][2], …
- Example: if 'a' is an int array, and B stores 4 int's per cache block (i.e. B=16 bytes), then
  - accessing a[0][0] will cause a cache miss, pulling in a[0][0], a[0][1], a[0][2] and a[0][3] from memory
  - When a[0][1], a[0][2] and a[0][3] are next accessed sequentially by the loop, they're already in cache!
  - So for every four accesses to the array a[i][j], we get 1 miss and 3 hits, for a miss rate of ¼=25%

Writing Cache Friendly Code

Repeated references to variables are good (temporal locality)

Stride-1 reference patterns are good (spatial locality)

Examples:

- **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%

- a[0][0] causes a cache miss, which brings in 4 words a[0][0] – a[0][3]
- a[0][1] is a cache hit, as is a[0][2] and a[0][3]
- a[0][4] causes a cache miss, bring in a[0][4]-a[0][7]
- Cache hits on a[0][5]-a[0][7]
- Etc…

Stride-N, Caching & Spatial Locality

Stepping through rows in one column: (Stride-N)

- `for (i = 0; i < n; i++)`
  `sum += a[i][0];`
- accesses distant elements
- no spatial locality!
  - When a[0][0] is accessed, there's a miss, so pull in a[0][0], a[0][1], a[0][2], a[0][3].
  - Next a[1][0] is accessed but it's not in the cache! So a miss. Fetch a[1][0], a[1][1], a[1][2], and a[1][3].
  - Next a[2][0] is accessed, but it's not in the cache!
  - Each access a[0][0], a[1][0], a[2][0], … causes a cache miss
  - So the miss rate = 1 (i.e. 100%)
- So penalty for a Stride-N access pattern is huge – may have to go to memory every time, suffering a large latency hit (~100X)

Writing Cache Friendly Code (2)

Repeated references to variables are good (temporal locality)

Stride-N reference patterns are bad (poor spatial locality)

Examples:

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

```
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;
}
```

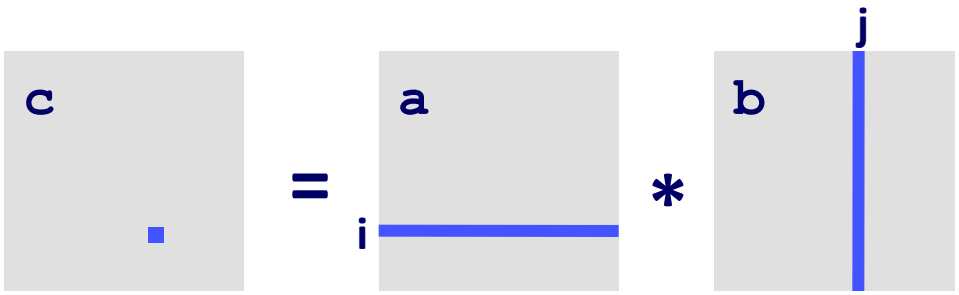
Miss rate = 100%

- a[0][0] causes a cache miss, which brings in 4 words a[0][0] – a[0][3]
- a[1][0] is a cache miss, brings in a[1][0] – a[1][3]
- a[2][0] is a cache miss, brings in a[2][0] – a[2][3]
- Etc…

Matrix Multiplication Example

```
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][j] += a[i][k] * b[k][j];
}
```



Description:

- Multiply two N x N matrices
- $O(N^3)$ total operations
  - Typically 3 loops
- Accesses
  - N reads per source element
  - N values summed per destination
    - » but may be able to hold in register

Matrix Multiplication Example (2)

```
/* 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*

Major 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

Matrix Multiplication Example (2)

```
/* 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;
  }
}
```

Assume in the following analysis:

Line size = 32B (big enough for 4 64-bit words)

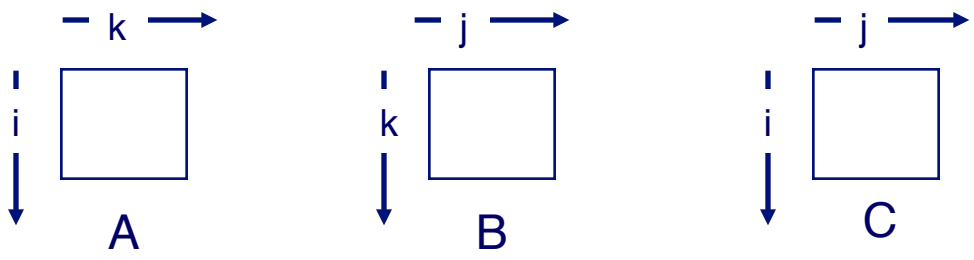
Matrix dimension (N) is very large

Approximate 1/N as 0.0

Cache is not even big enough to hold multiple rows

Analysis Method:

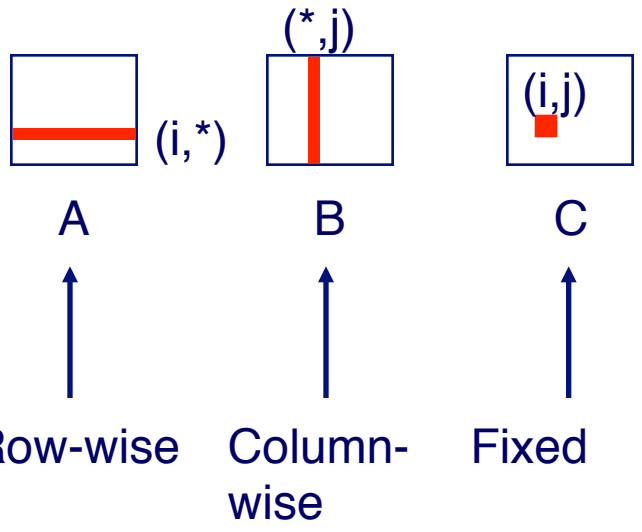
- **Look at access pattern of inner loop**



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;
  }
}
```

Inner loop:



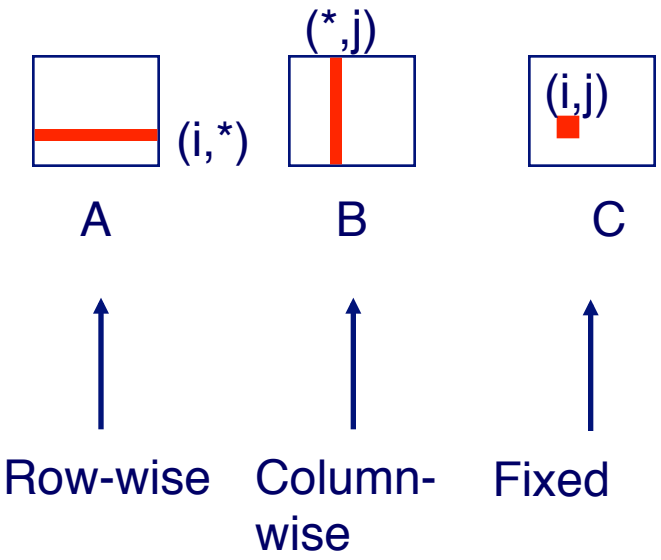
Misses per Inner Loop Iteration:

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
  }
}
```

Inner loop:

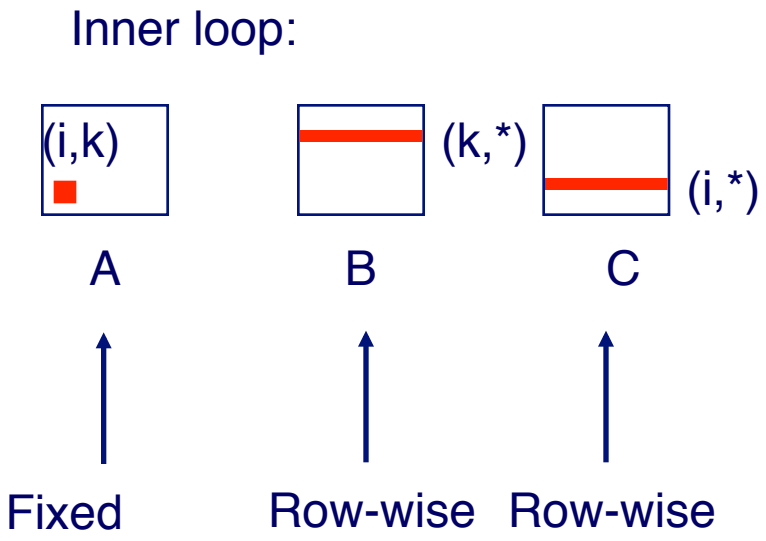


Misses per Inner Loop Iteration:

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];
  }
}
```

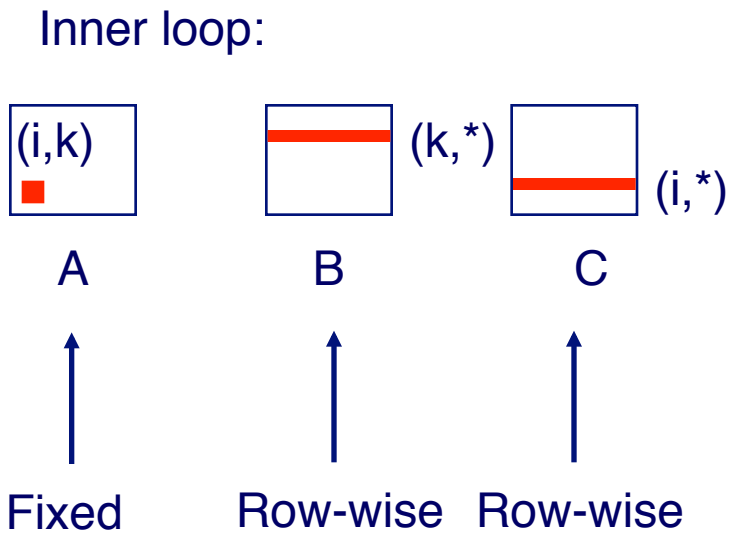


Misses per Inner Loop Iteration:

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];
  }
}
```

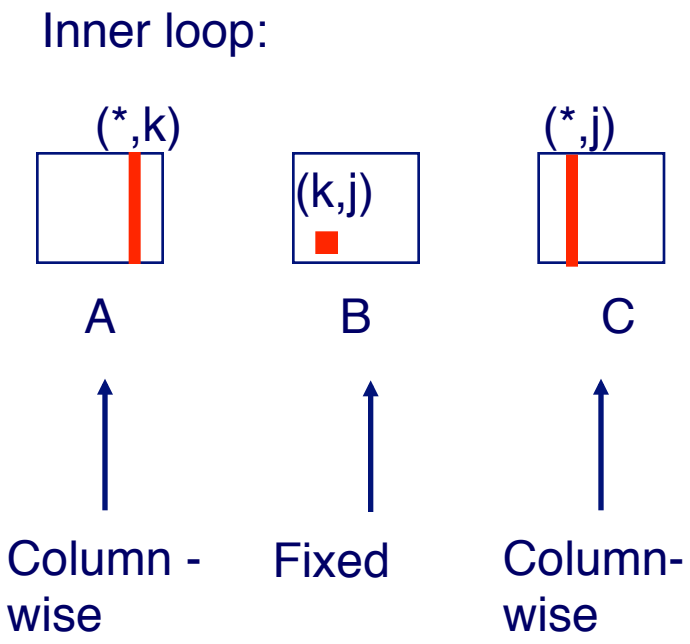


Misses per Inner Loop Iteration:

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;
  }
}
```

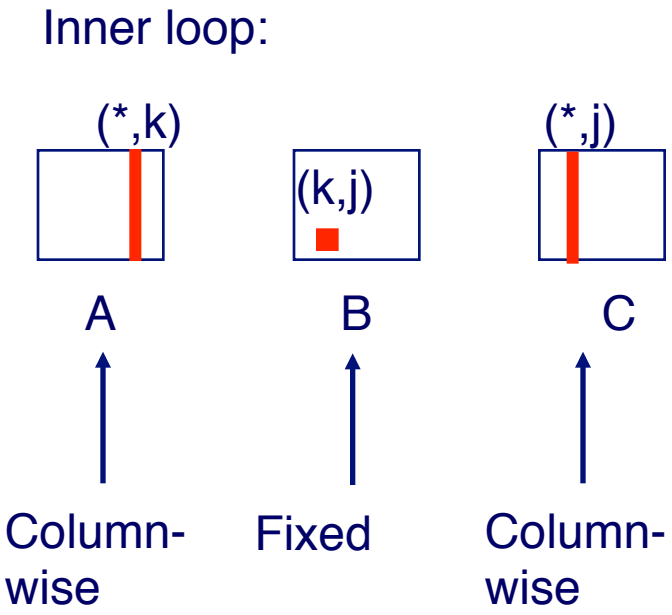


Misses per Inner Loop Iteration:

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**

```
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**
- Stride-1 for both inner arrays

```
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];
   }
}
```

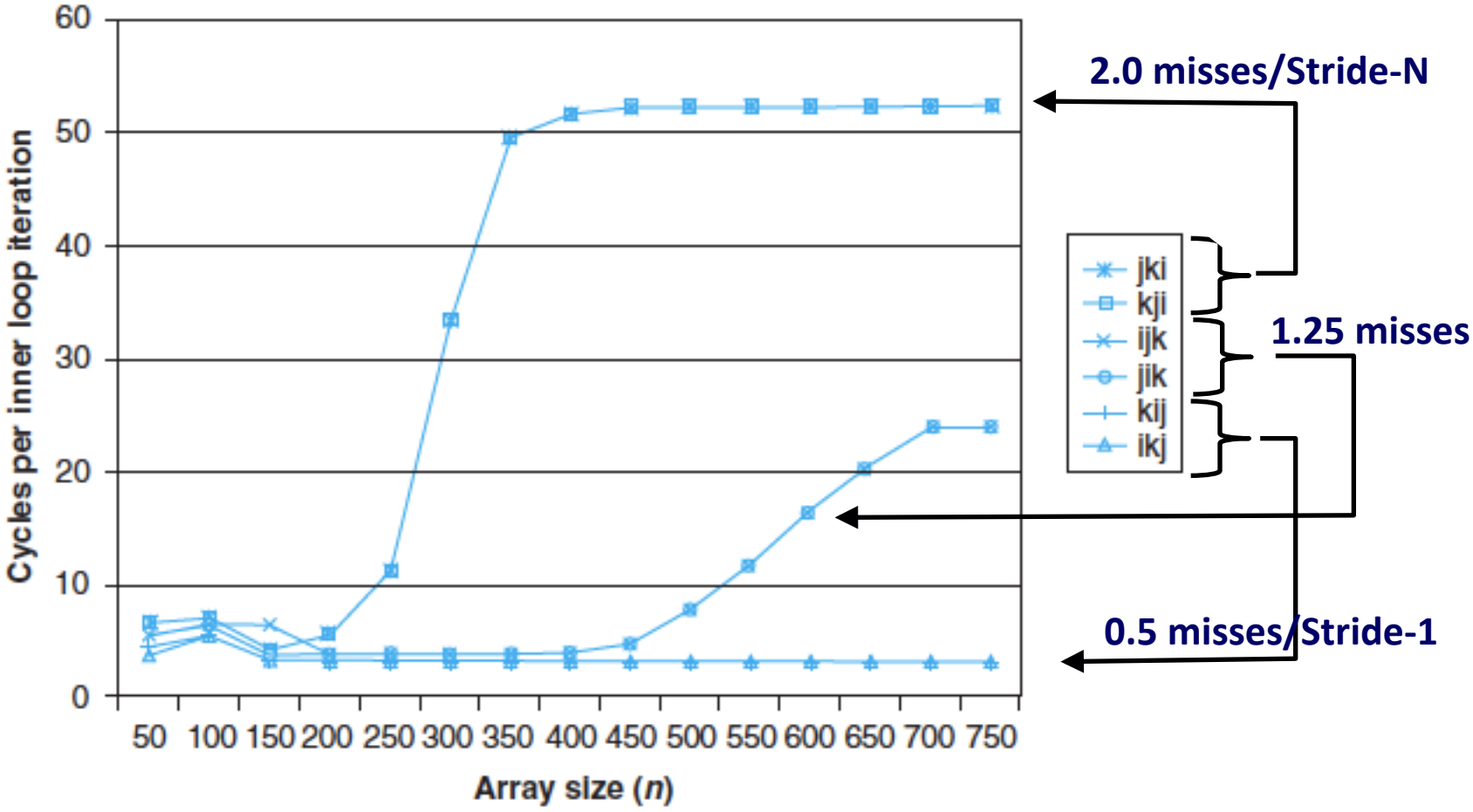
Cache-Friendly Code!

jki (& kji):
- 2 loads, 1 store
- misses/iter = **2.0**
- Stride-N for both inner arrays

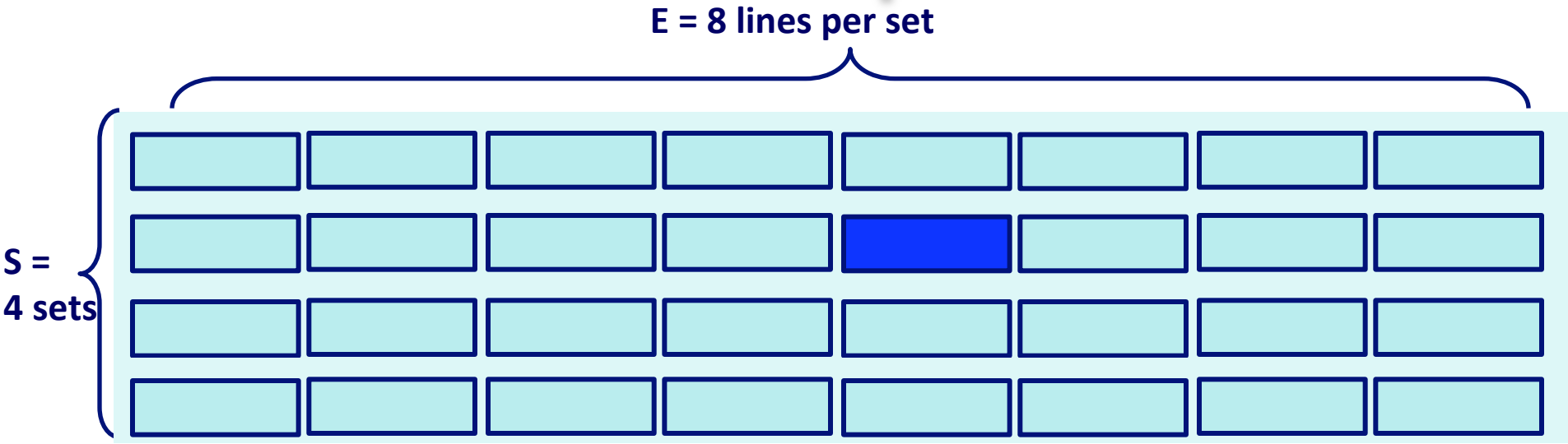
```
for (j=0; j<n; j++) {
   for (k=0; k<n; k++) {
      r = b[k][j];
      for (i=0; i<n; i++)
         c[i][j] += a[i][k] * r;
   }
}
```

Core i7 Matrix Multiply Performance

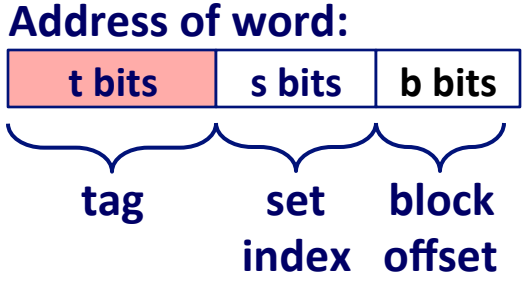
Actual inner loop performance = f(miss rate, # loads & stores)



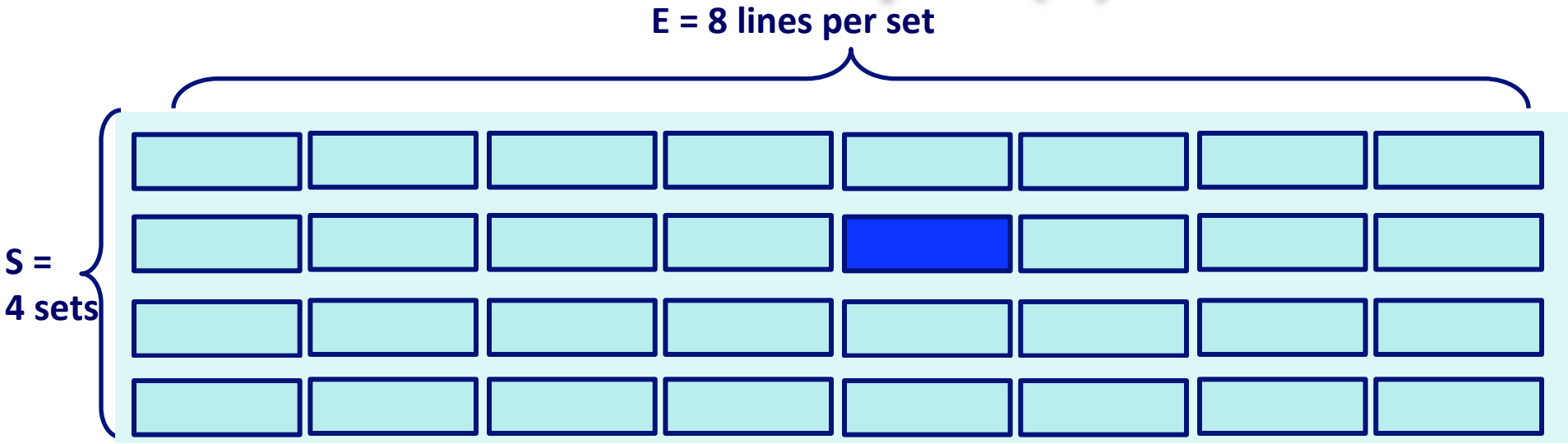
Cache Access Example



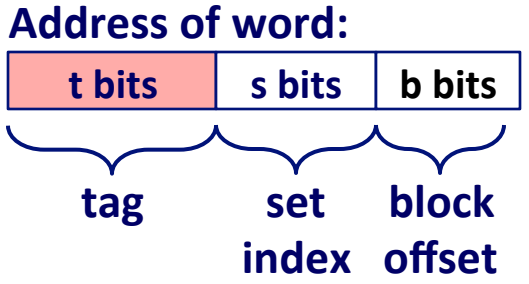
- *Suppose* cache stores 32-bit data or instructions = 4 bytes/block or line
- Each cached block is taken from a location in memory, so could use the cached block's full memory address as its identifier or tag.
- But since each block is 4 bytes long, then the memory addresses of blocks are unique only to a factor of 4
  - Lower 2 bits of address not needed in tagging a block. (b=2)



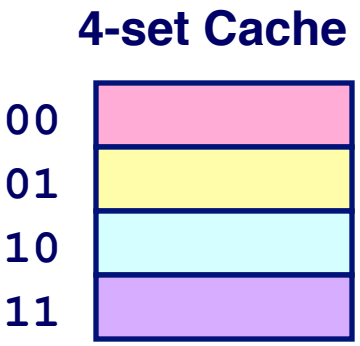
Cache Access Example (2)



- Of remaining 30 bits, suppose the hash function separates these into 4 sets: the set or row that an address belongs to in the cache = hash h(top 30 bits of address) = *(top 30 bits of address) mod 4*
  - This hashes an address into 1 of 4 hash buckets/rows or sets, as shown above.
  - Thus, the next 2 lower order bits are used to identify a row (s=2)
- Within a row, tag size t = 32 - (2+2) = 28 bits
  - Tag of cache block = top 28 bits (t=28)
  - Match these tags with the input address



Why Use Middle Bits as Index/Hash?

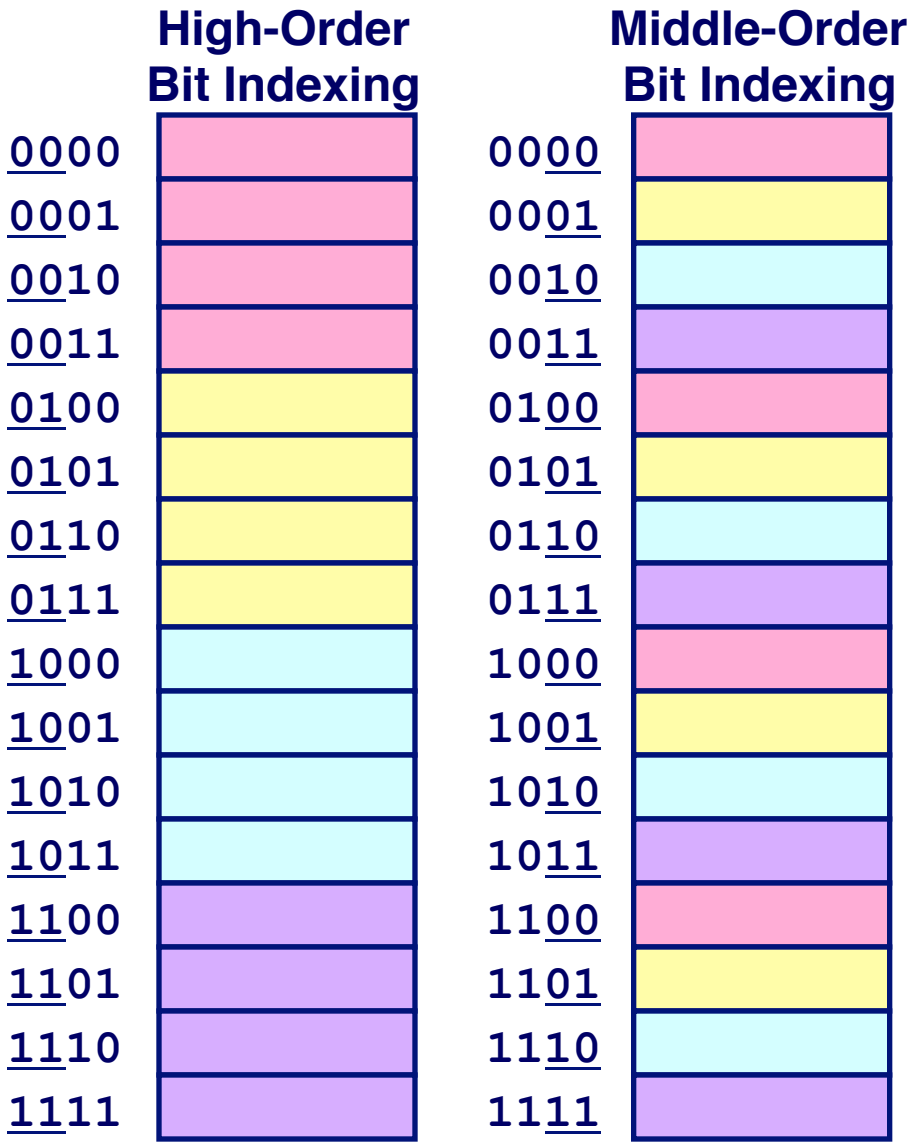


High-Order Bit Indexing

- Adjacent memory lines would map to same cache entry
- Poor use of spatial locality

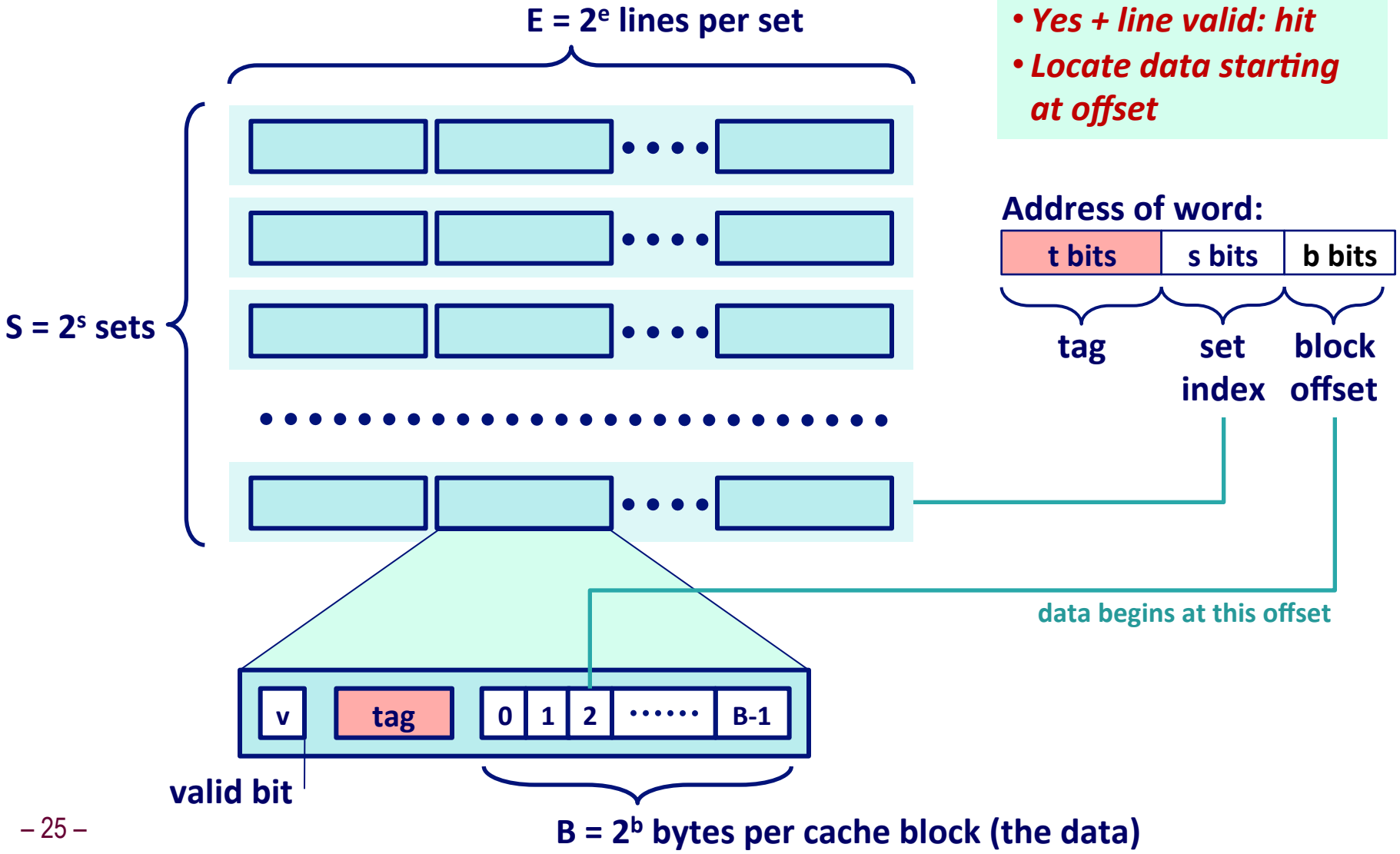
Middle-Order Bit Indexing

- Consecutive memory lines map to different cache lines
- Can hold C-byte region of address space in cache at one time



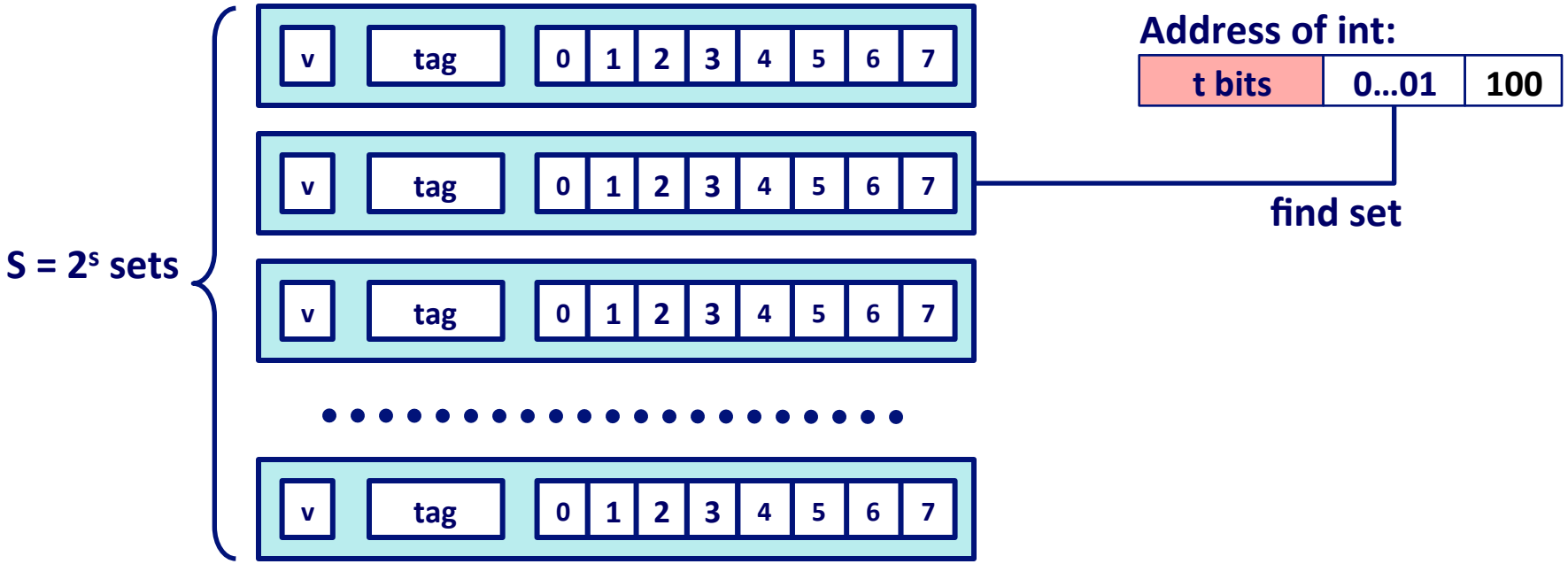
Cache Read

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



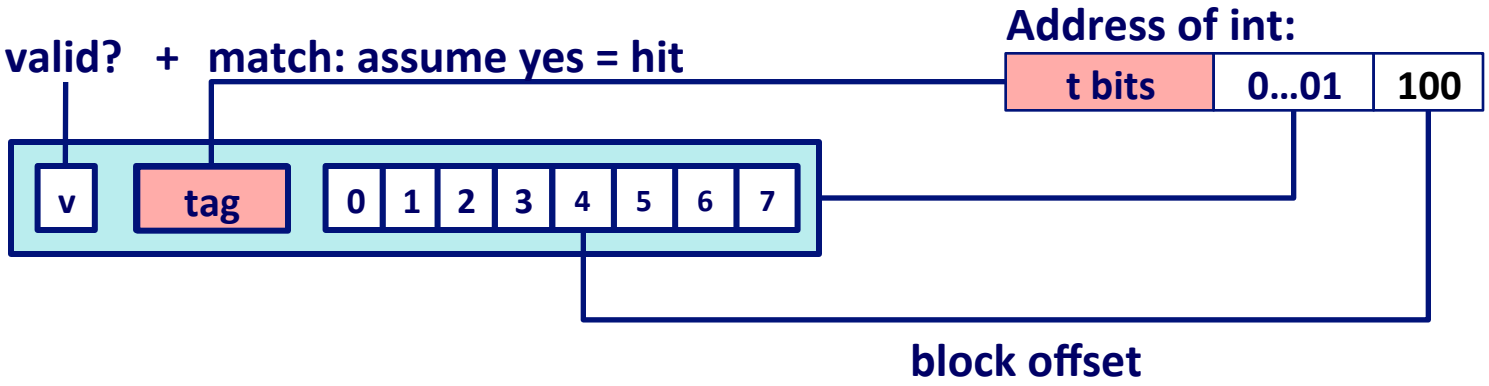
Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set
Assume: cache block size 8 bytes



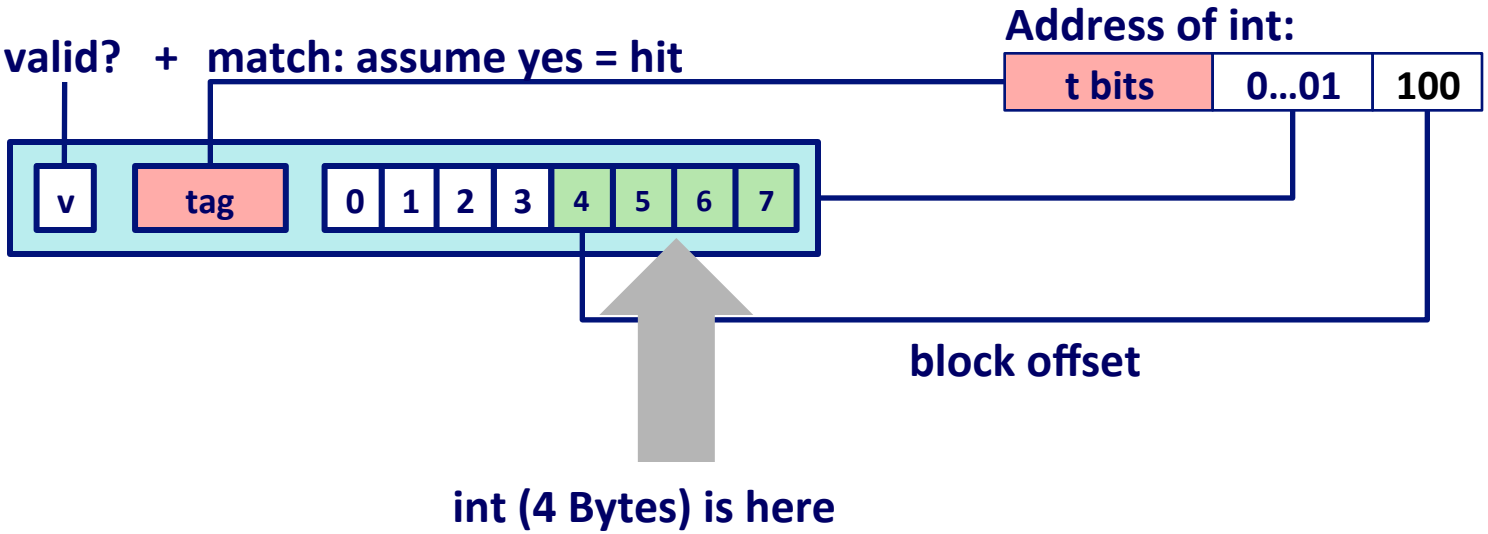
Example: Direct Mapped Cache (E = 1)

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 bytes of addresses, B=2 bytes/block,
S=4 sets, E=1 entry/set

Address trace (reads):
0 [0000_2], 1 [0001_2], 13 [1101_2], 8 [1000_2], 0 [0000_2]

0 [0000_2] *(miss)*

(1)

v	tag	data
1	0	M[0-1]

hit

13 [1101_2] *(miss)*

(3)

v	tag	data
1	0	M[0-1]
1	1	M[12-13]

8 [1000_2] *(miss)*

(4)

v	tag	data
1	1	M[8-9]
1	1	M[12-13]

0 [0000_2] *(miss)*

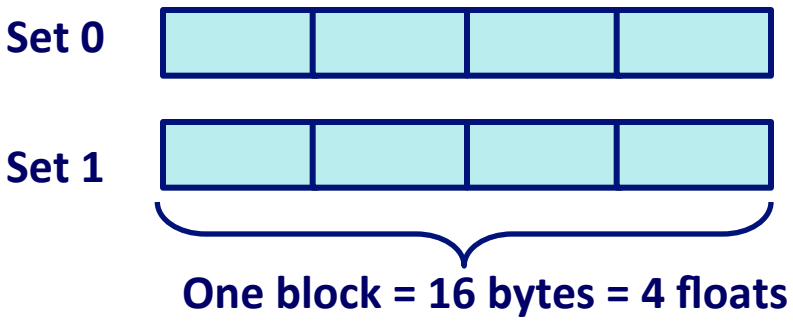
(5)

v	tag	data
1	0	M[0-1]
1	1	M[12-13]

Direct-Mapped Cache Performance Problem

```
float dotprod(float x[8], float y[8])
{
    int i;
    float sum = 0;

    for (i = 0; i < 8; i++)
        sum += x[i]*y[i];
    return sum;
}
```



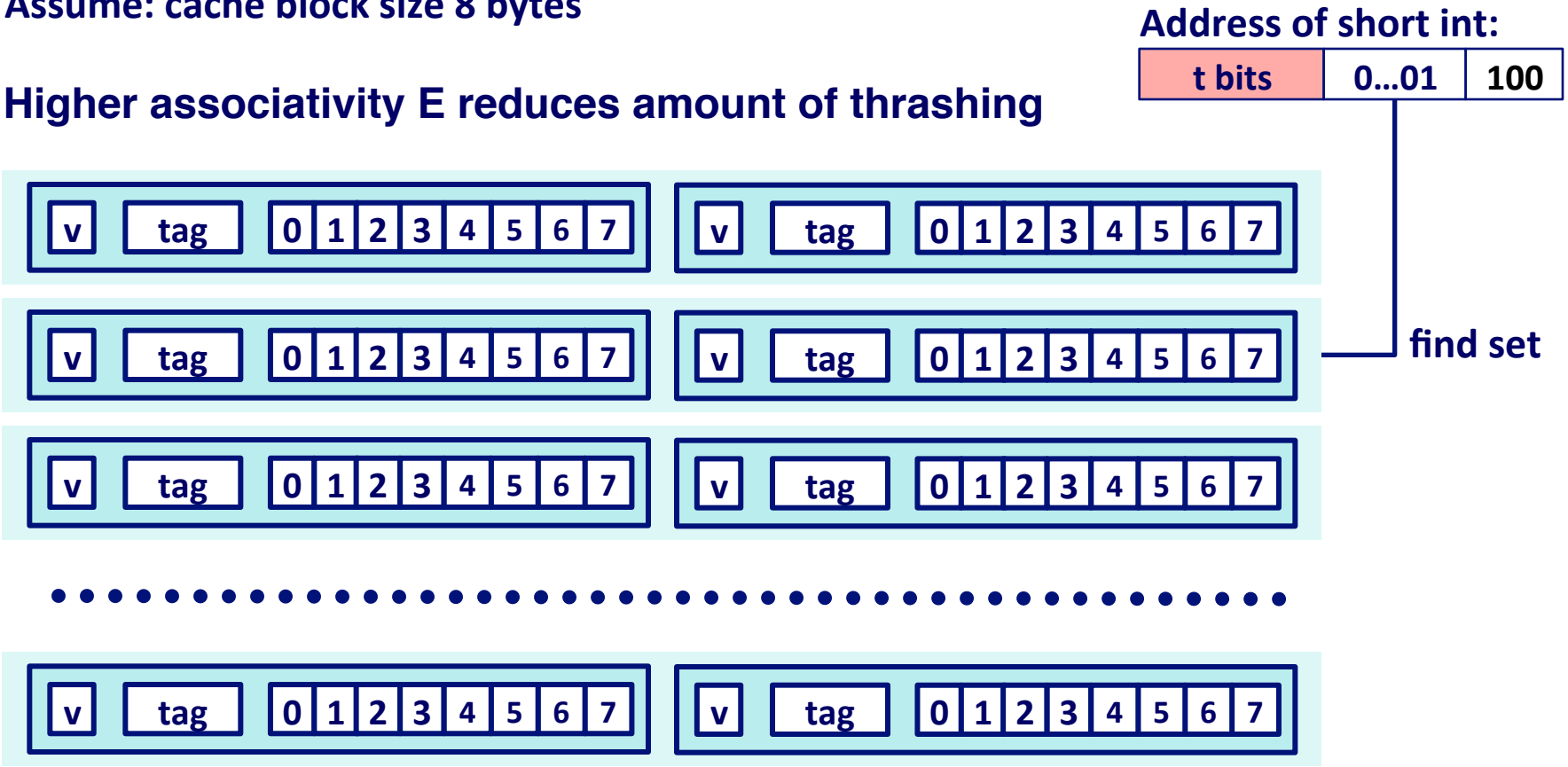
- Assume x[] is loaded into 32 bytes of contiguous memory at address A divisible by 32, and y[] starts immediately after x[]
- Each x[i] and y[i] maps to the identical cache set
- On first iteration, x[0] is a cache miss, so pull in x[0]-x[3].
- Next, need y[0], also a cache miss, so pull in y[0]-y[3] – replaces x[0]-x[3]!
- x[1] misses, overwriting y[0]-y[3], etc…, so we get *thrashing*

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

E = 2: Two lines per set

Assume: cache block size 8 bytes

Higher associativity E reduces amount of thrashing

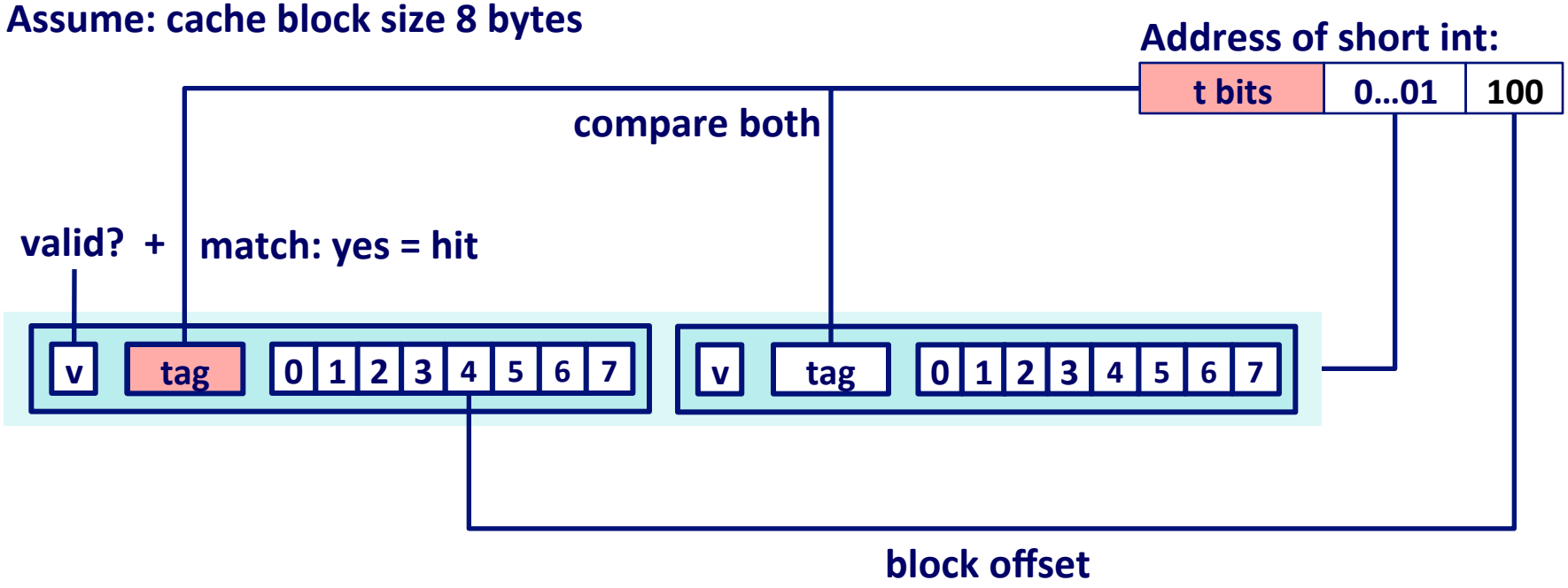


But note: if your data access pattern maps to the same row of a cache, and rotates through 2*E data items, then you'll still get thrashing

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

E = 2: Two lines per set

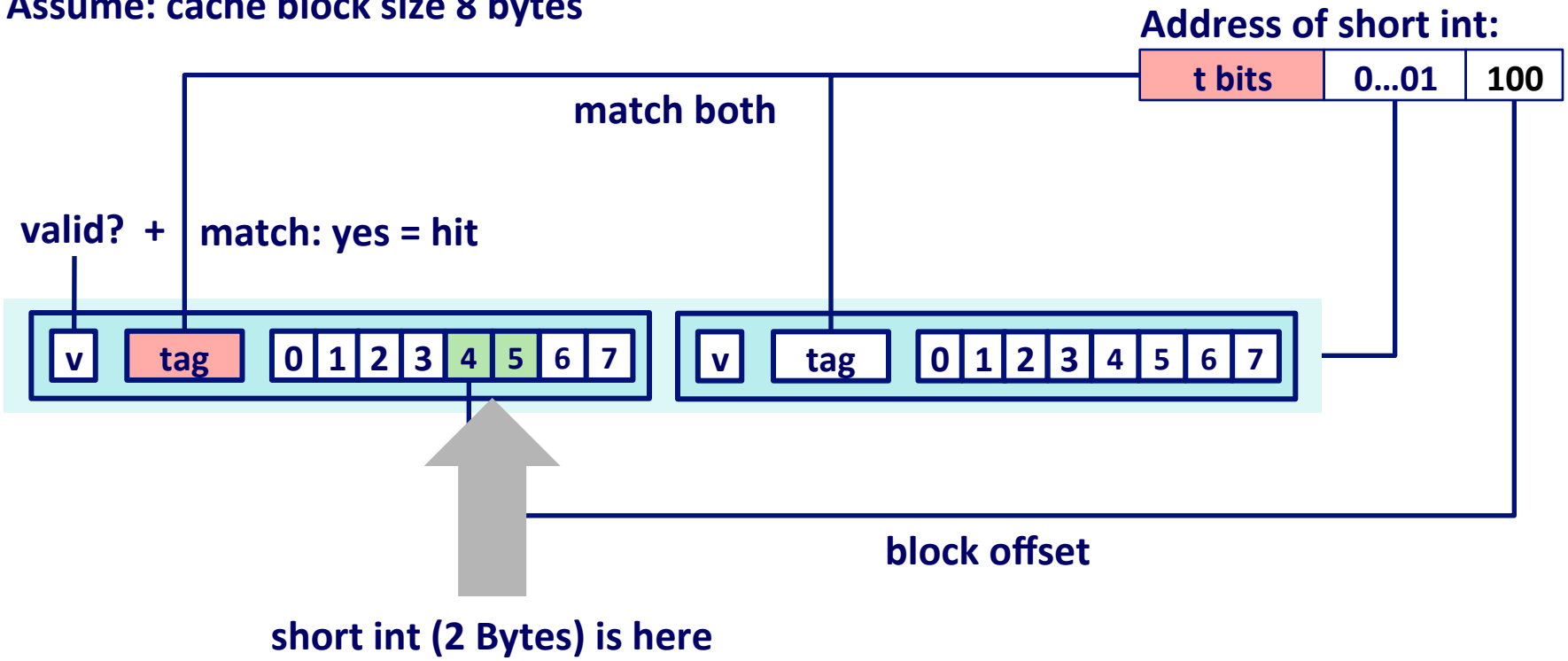
Assume: cache block size 8 bytes



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

E = 2: Two lines per set

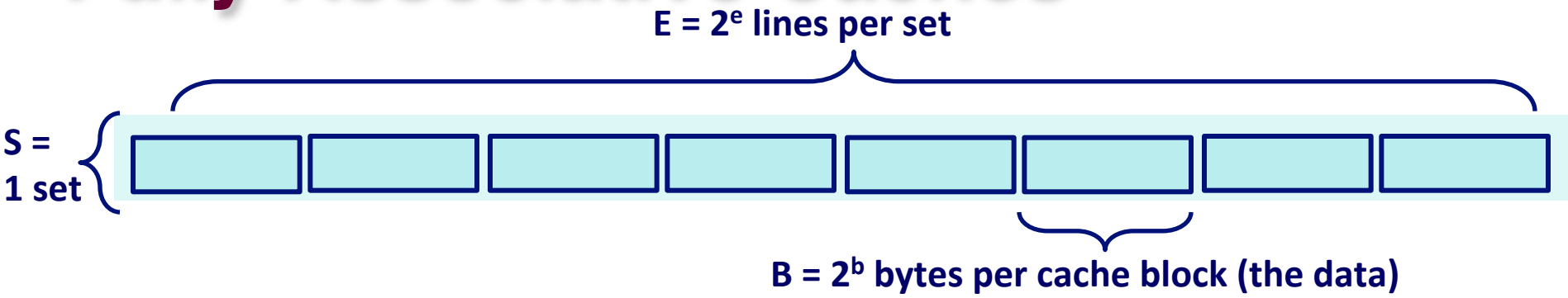
Assume: cache block size 8 bytes



No match and full (both lines valid):

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

Fully Associative Caches



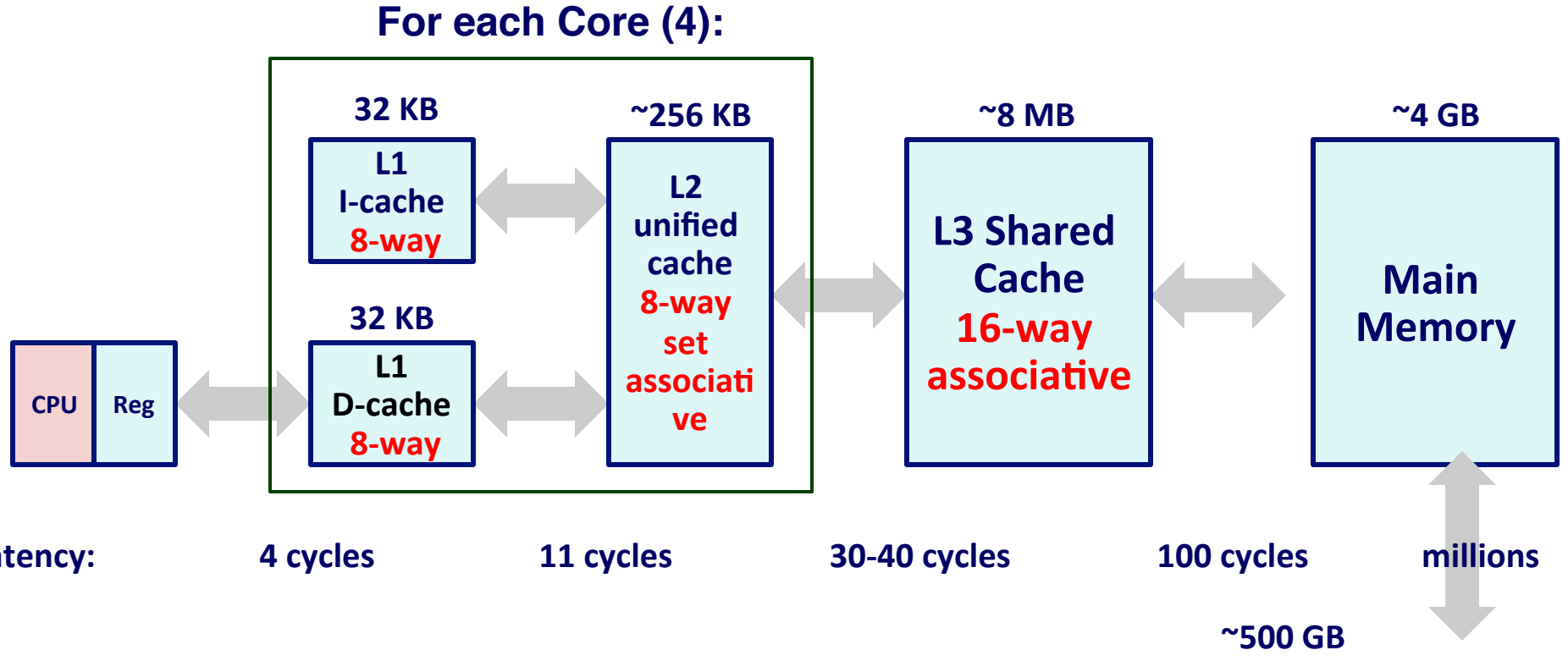
Consists of a single set, i.e. S=1, and C = E*B, or E=C/B

- All the work is done in line matching and word selection
- This works the same as for earlier caches
- This is expensive, requiring more parallel circuitry for matching
- This is reserved for special caches, e.g. TLBs

Memory Hierarchy: Intel Core i7

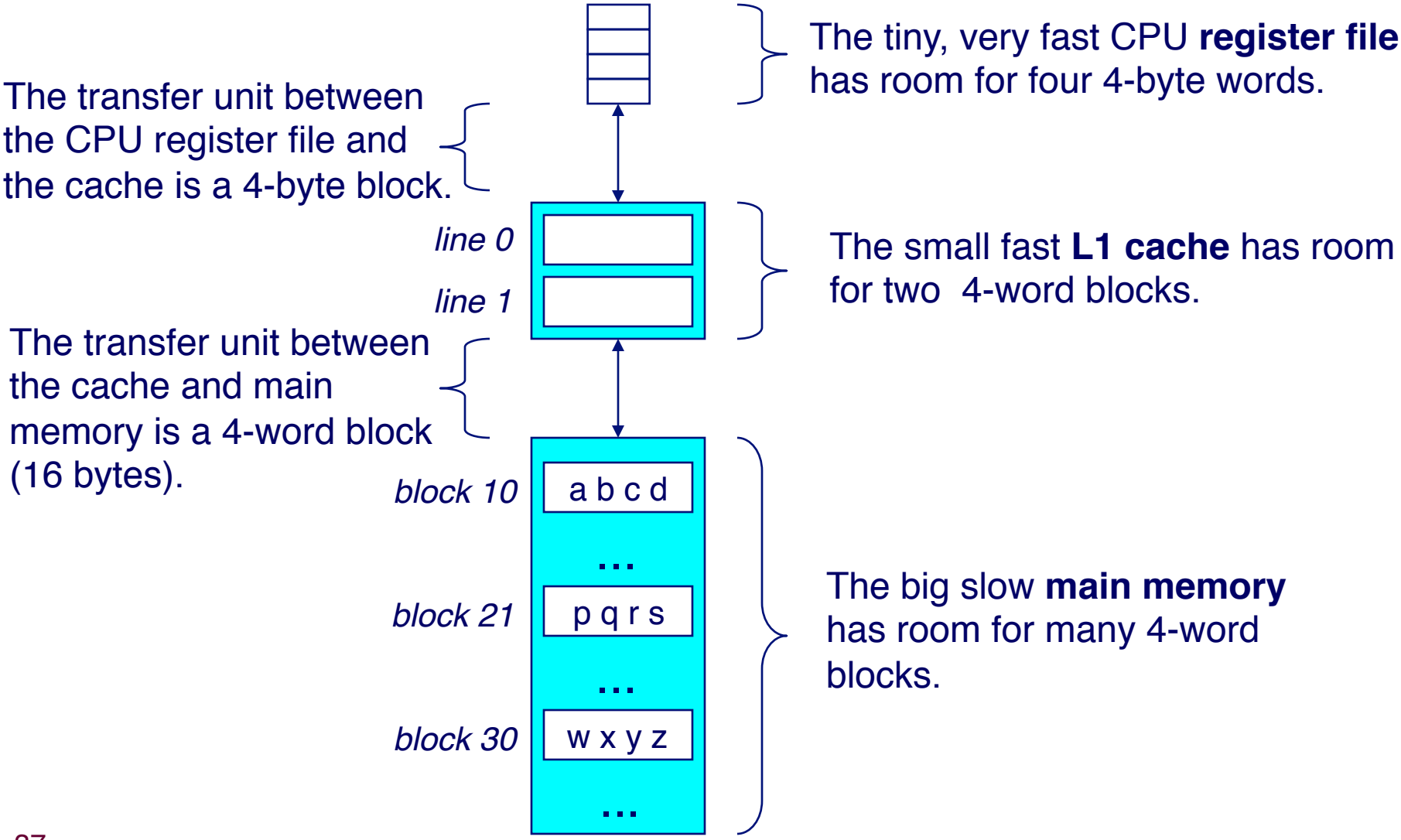
Not drawn to scale

L1/L2/L3 caches: 64 B blocks



Supplementary Slides

Inserting an L1 Cache Between the CPU and Main Memory

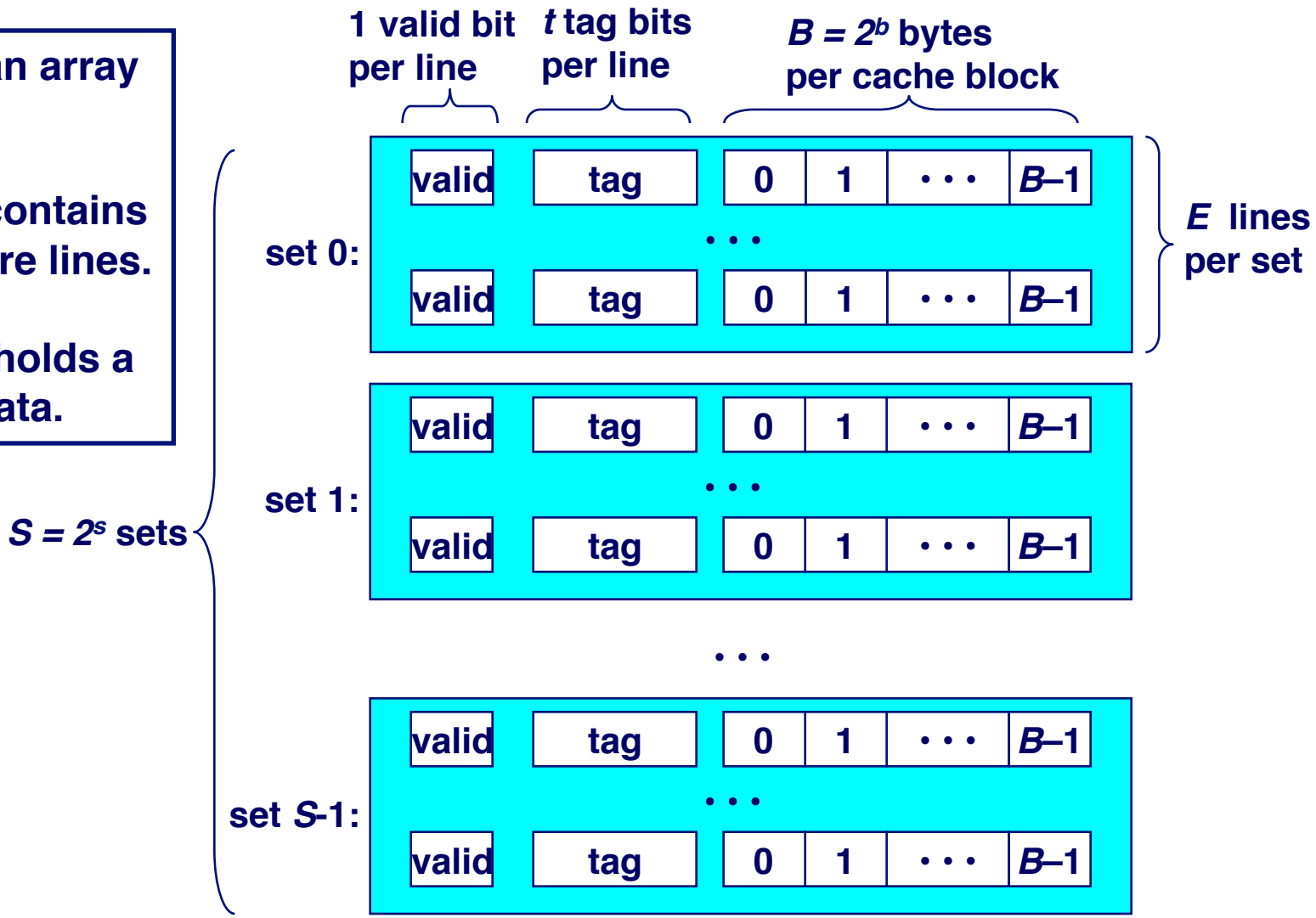


General Org of a Cache Memory

Cache is an array of sets.

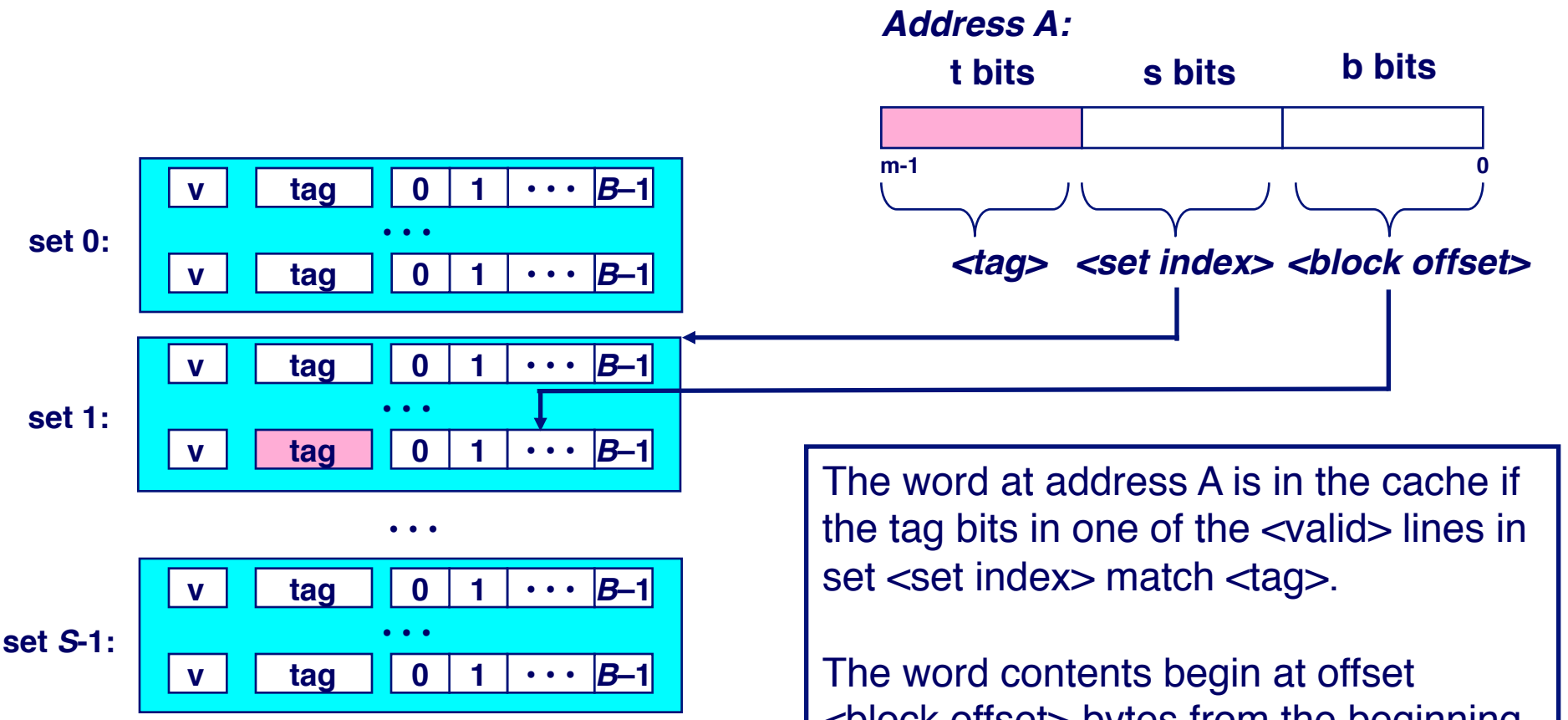
Each set contains one or more lines.

Each line holds a block of data.



Cache size: $C = B \times E \times S$ data bytes

Addressing Caches



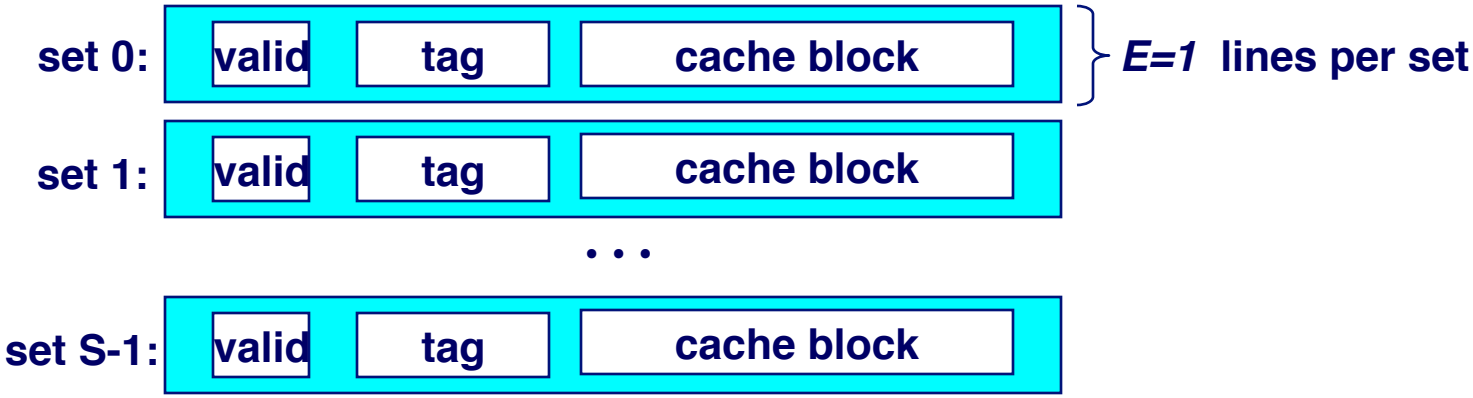
The word at address A is in the cache if the tag bits in one of the <valid> lines in set <set index> match <tag>.

The word contents begin at offset <block offset> bytes from the beginning of the block.

Direct-Mapped Cache

Simplest kind of cache

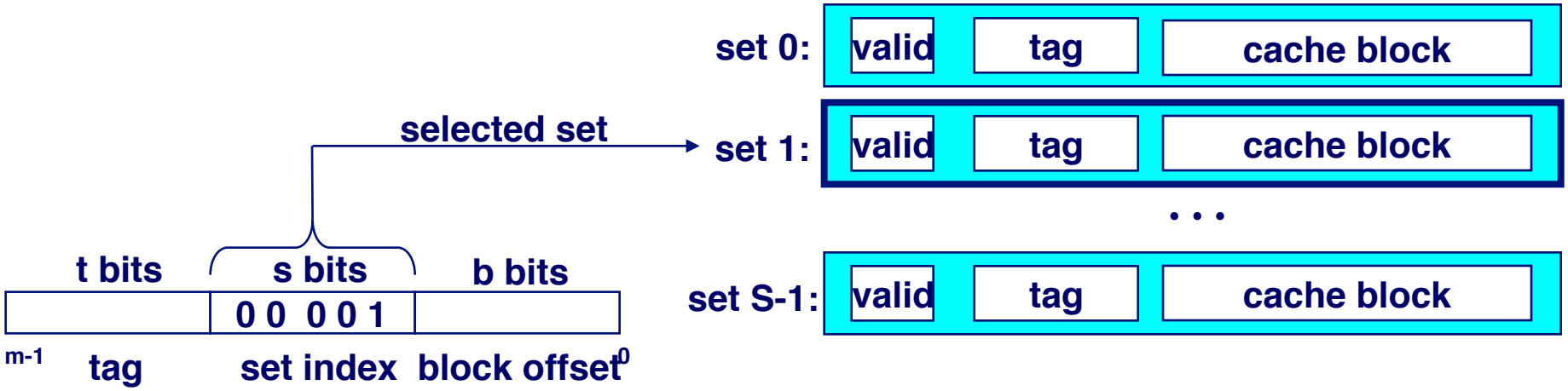
Characterized by exactly one line per set.



Accessing Direct-Mapped Caches

Set selection

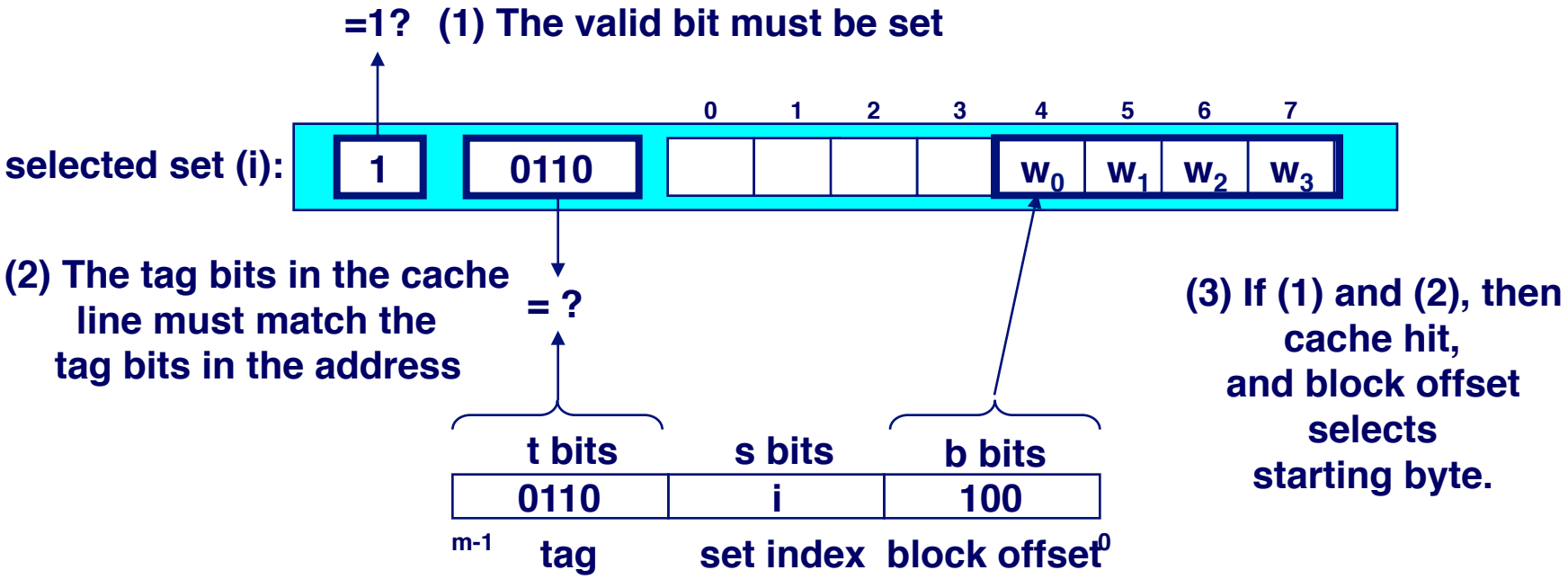
- Use the set index bits to determine the set of interest.



Accessing Direct-Mapped Caches

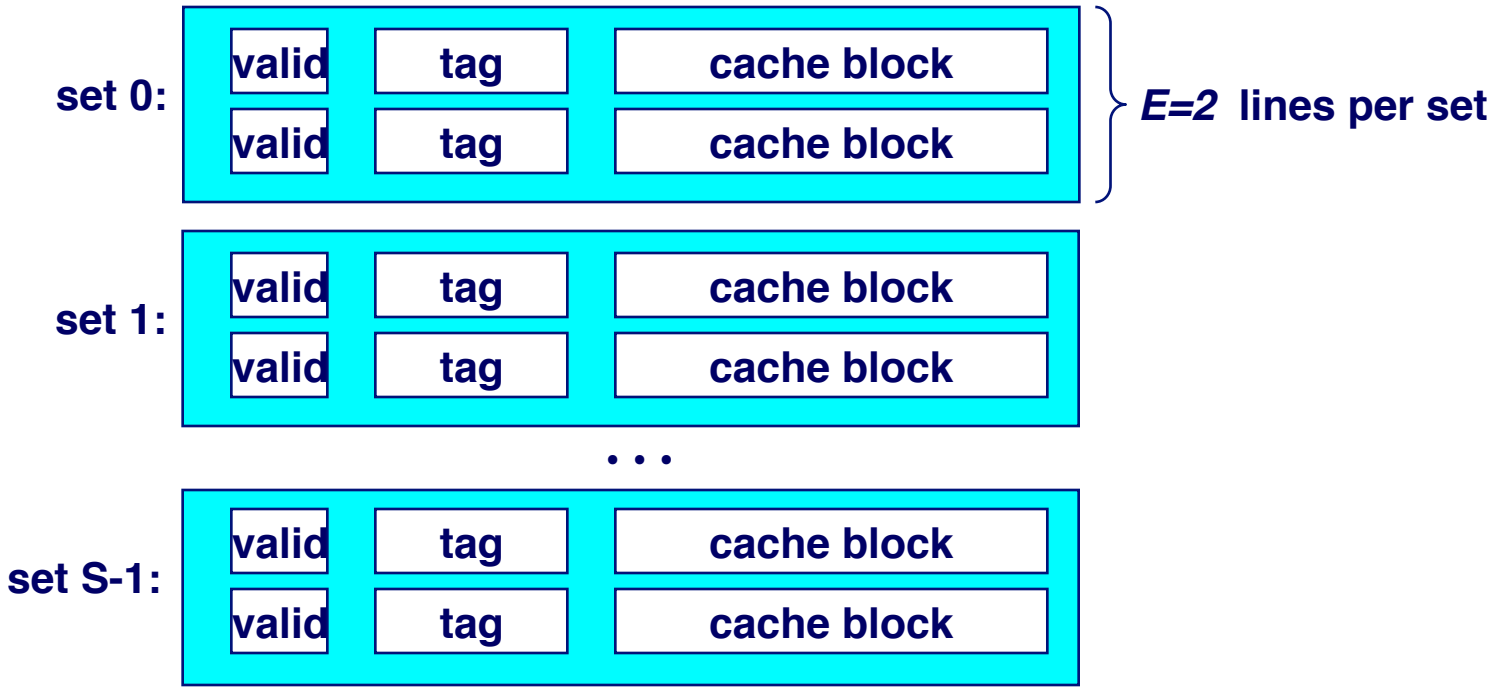
Line matching and word selection

- **Line matching**: Find a valid line in the selected set with a matching tag
- **Word selection**: Then extract the word



Set Associative Caches

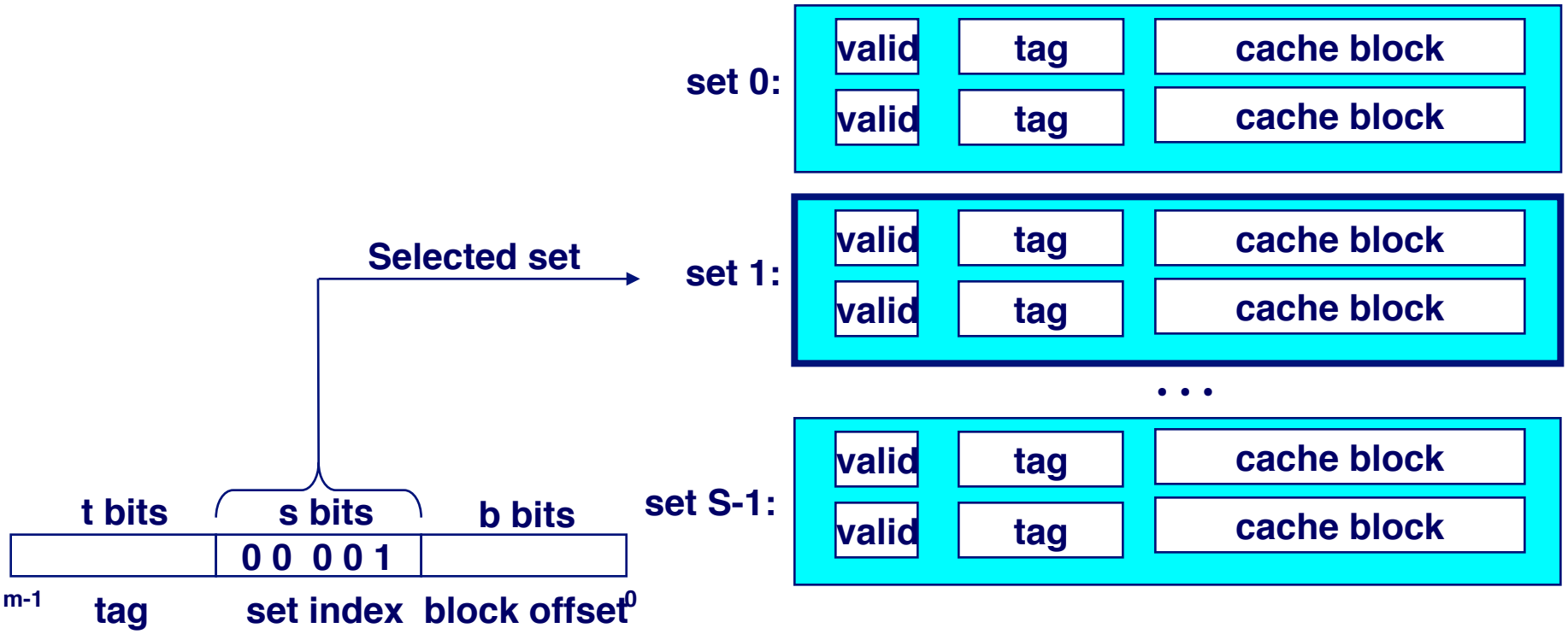
Characterized by more than one line per set



Accessing Set Associative Caches

Set selection

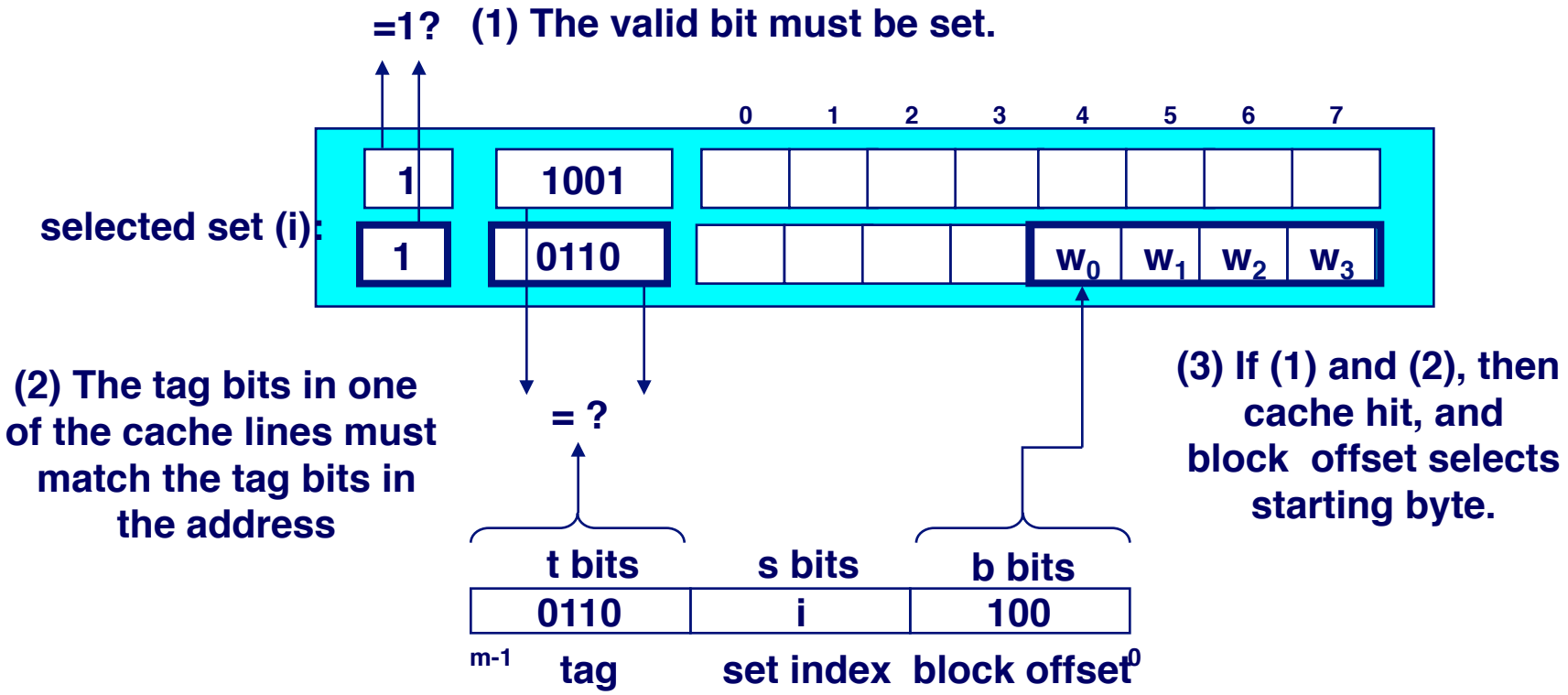
- identical to direct-mapped cache



Accessing Set Associative Caches

Line matching and word selection

- must compare the tag in each valid line in the selected set.



Software Caches are More Flexible

Examples

- File system buffer caches, web browser caches, etc.

Some design differences

- Almost always fully associative
  - so, no placement restrictions
  - index structures like hash tables are common
- Often use complex replacement policies
  - misses are very expensive when disk or network involved
  - worth thousands of cycles to avoid them
- Not necessarily constrained to single “block” transfers
  - may fetch or write-back in larger units, opportunistically