Memory hierarchies: caches and their impact on the running time

Irene Finocchi

Dept. of Computer and Science

Sapienza University of Rome

A happy coincidence

A fundamental property of hardware

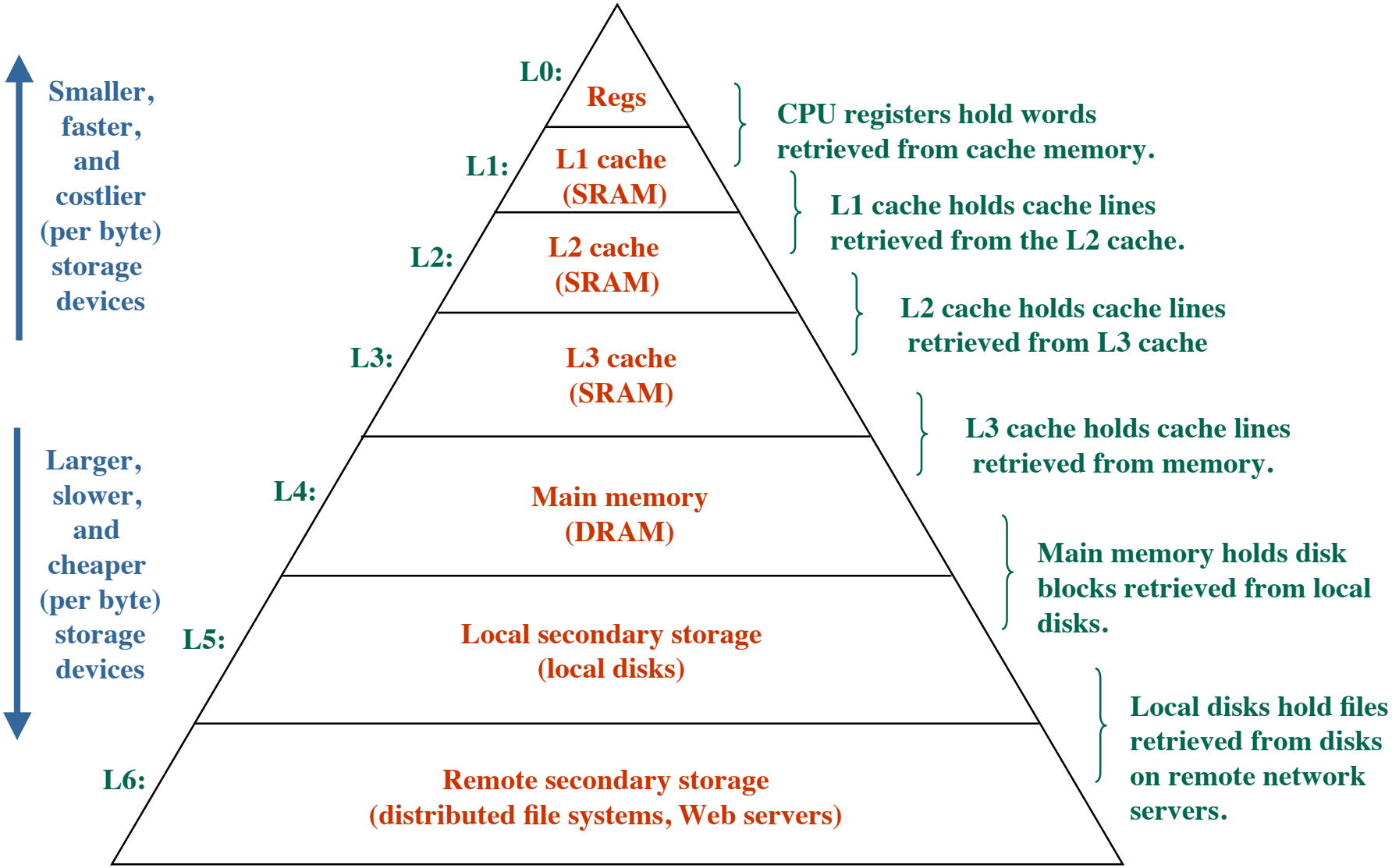
Different storage technologies have widely different access times. Faster technologies cost more per byte than slower ones and have less capacity. The gap between CPU and main memory speed is widening.

A fundamental property of software

Well-written programs tend to exhibit good locality

The two properties complement each other!

The memory hierarchy



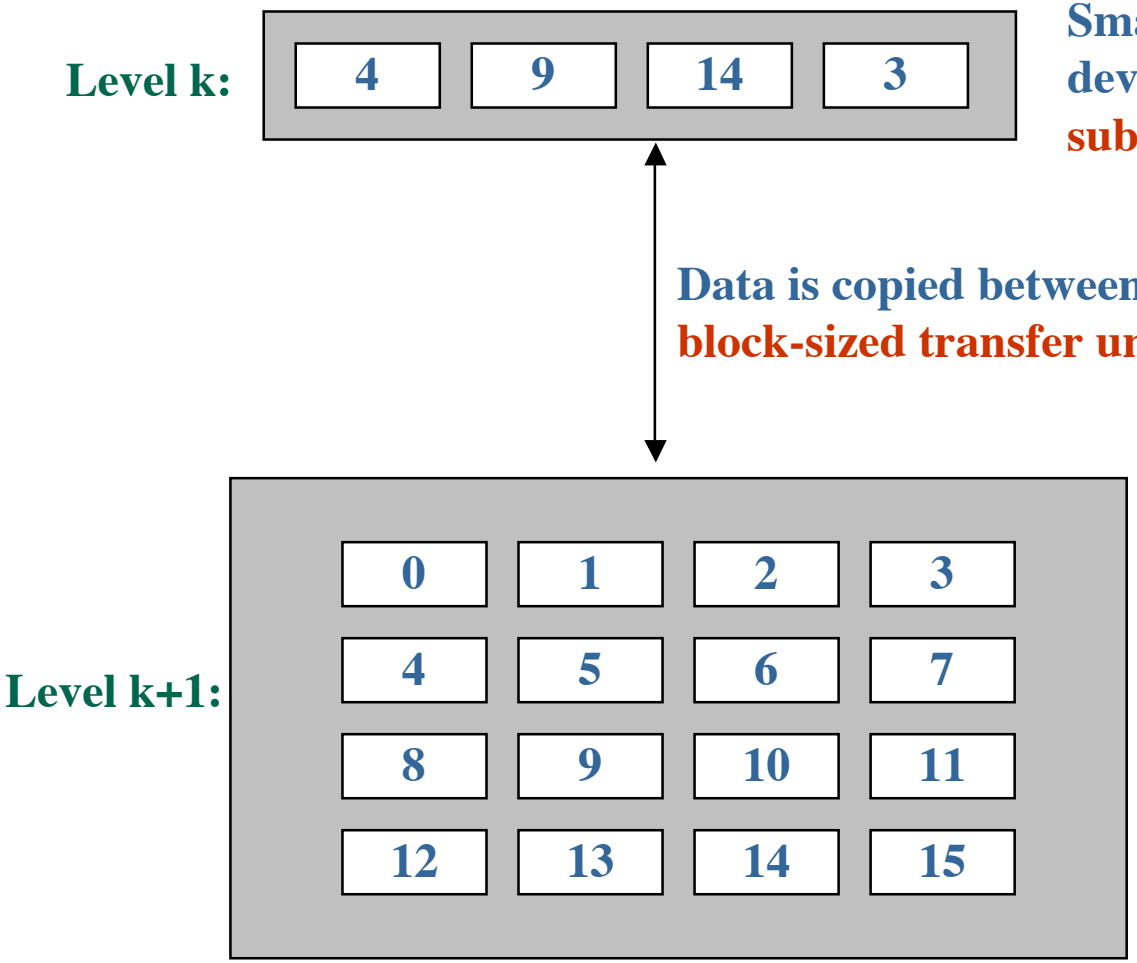
Caching in the memory hierarchy

Cache:

small, fast storage device that acts as a staging area for the data objects stored in a larger, slower device

For each k, the faster and smaller storage at level k serves as a cache for the larger and slower storage device at level k+1

Basic principle of caching



Smaller, faster, more expensive device at level k caches a **subset of the blocks from level k+1**

Data is copied between levels in **block-sized transfer units**

Larger, slower, cheaper storage device at level k+1 is **partitioned into blocks**.

Who is responsible for caching?

Cache type	What cached	Managed by
CPU registers	4 or 8 byte words	Compiler
L1 cache	64-byte block	Hardware
L2 cache	64-byte block	Hardware
L3 cache	64-byte block	Hardware
Virtual memory	4KB page	Hardware + OS

Can we exploit caches?

- Same principles seen in external memory model: increase spatial/temporal locality
- But caches in the memory hierarchy far more complex

1. Parameter choice?
2. Automatic hardware management: how can we control which data are cached?

Still, programmers can write cache-friendly code and caches can have a big impact

A simple experiment

Copy 2048 x 2048 array of 32-bit integers

`copyij:` copy by rows

`copyij:` copy by columns

Array copy: row-major order

Access by rows:

```
void copyij (int src[2048][2048],
             int dst[2048][2048])
{
  int i,j;
  for (i = 0; i < 2048; i++)
     for (j = 0; j < 2048; j++)
          dst[i][j] = src[i][j];
}
```

Array copy: column-major order

Access by columns:

```
void copyji (int src[2048][2048],
             int dst[2048][2048])
{
  int i,j;
  for (j = 0; j < 2048; j++)
     for (i = 0; i < 2048; i++)
          dst[i][j] = src[i][j];
}
```

Performance analysis

`copyij` and `copyji` differ only in data access patterns:

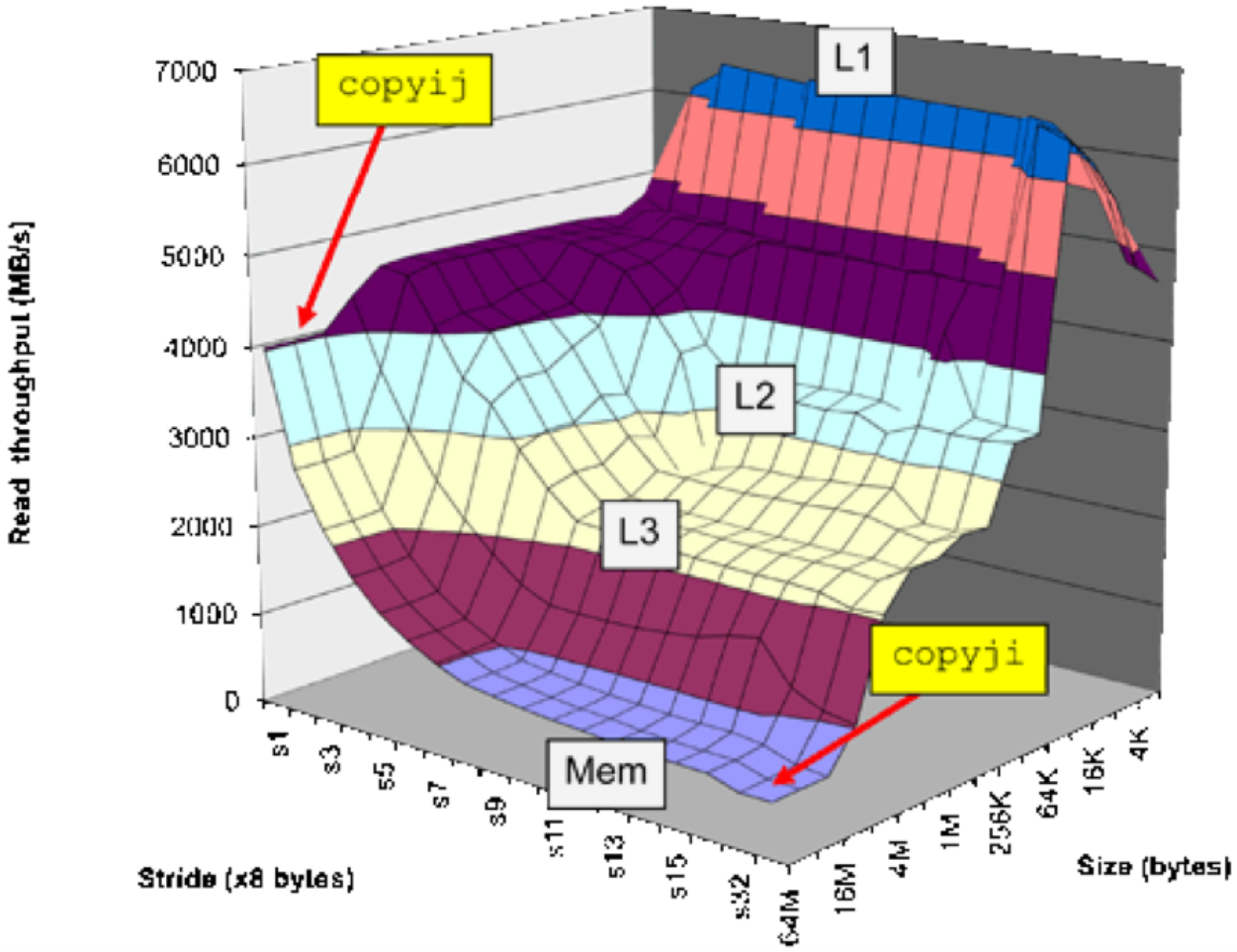
- `copyij` accesses by rows
- `copyji` accesses by columns

On a Intel Core i7 with 2.7 GHz:

- `copyij` takes 5.2 msec,
- `copyji` takes 162 msec (≈ 30x slower!)

`copyij` makes a better use of (spatial) locality

Is this due to memory access times?

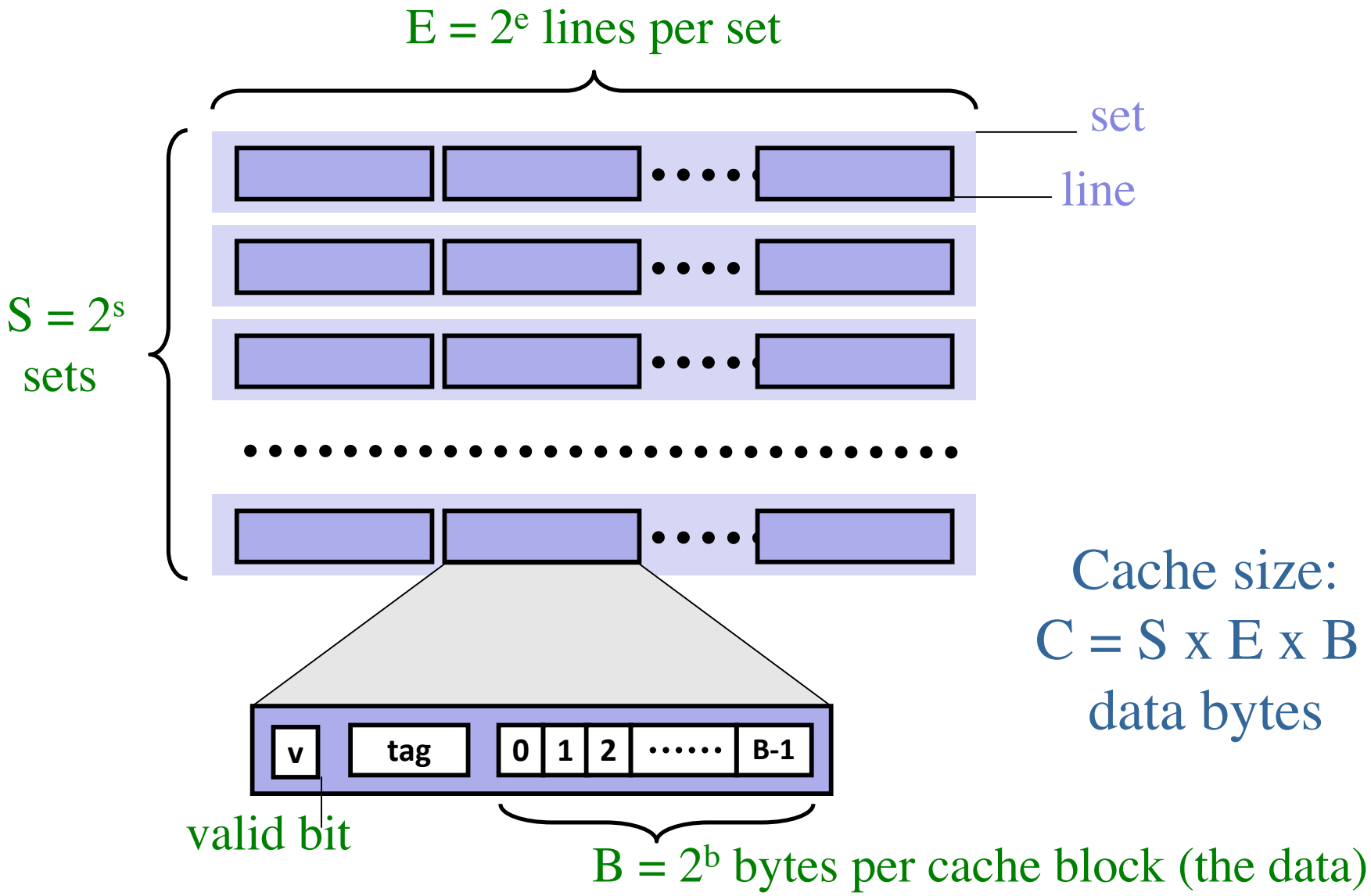


General cache organization

Caches automatically managed by hardware

Need to understand how caches work to exploit them

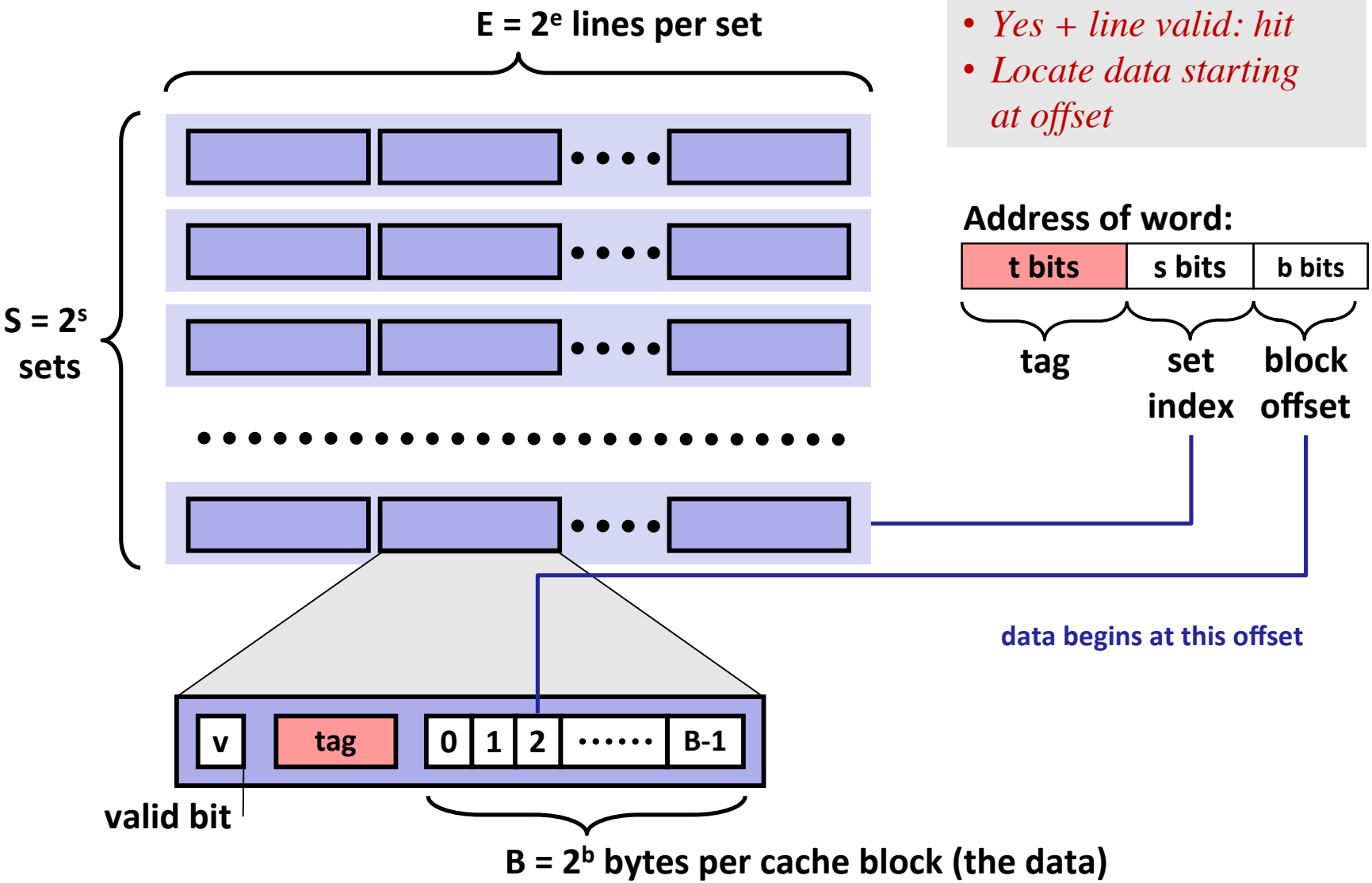
General cache organization (S, E, B)



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

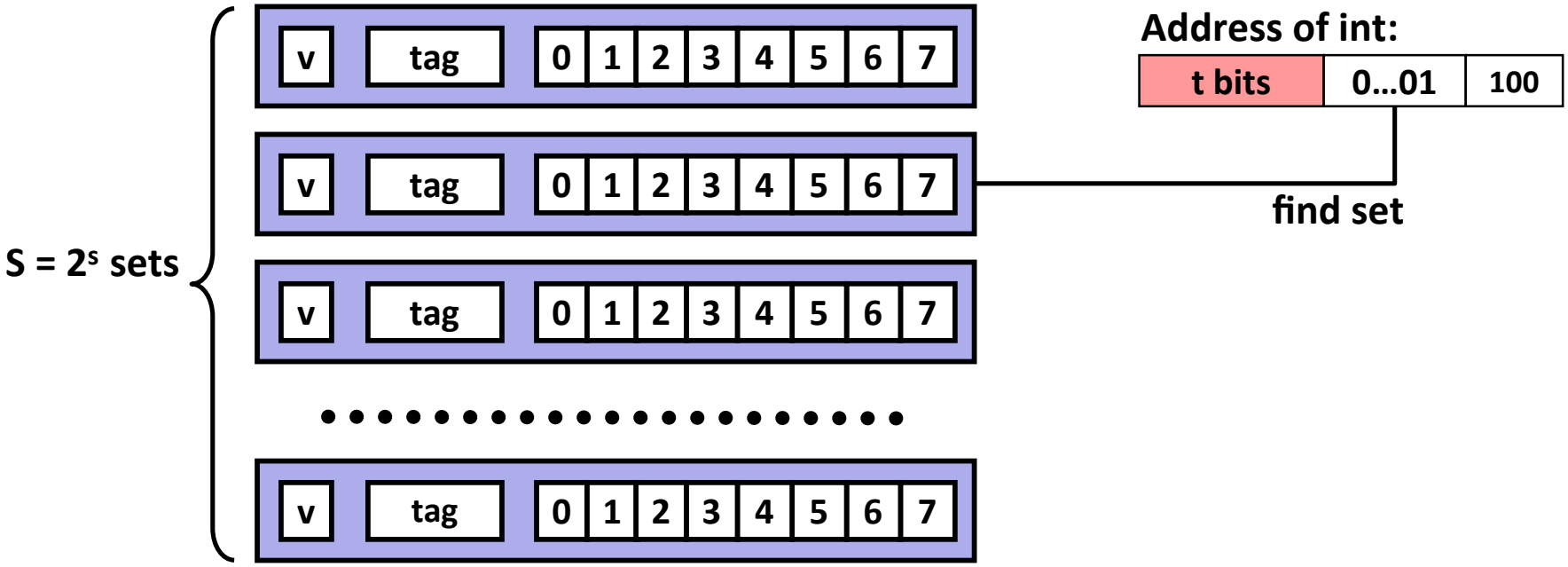
Cache read

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



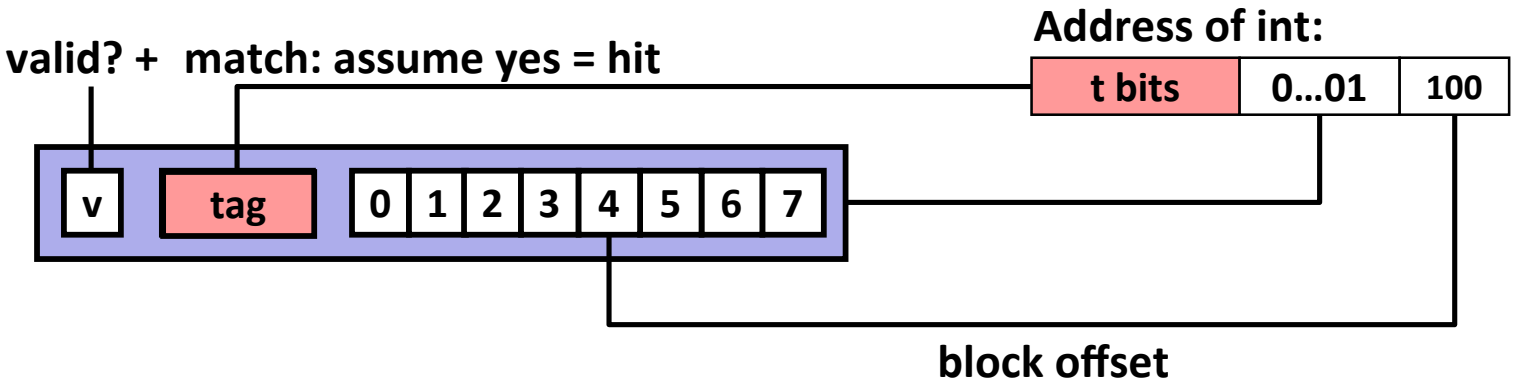
Example: direct mapped cache (E = 1)

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



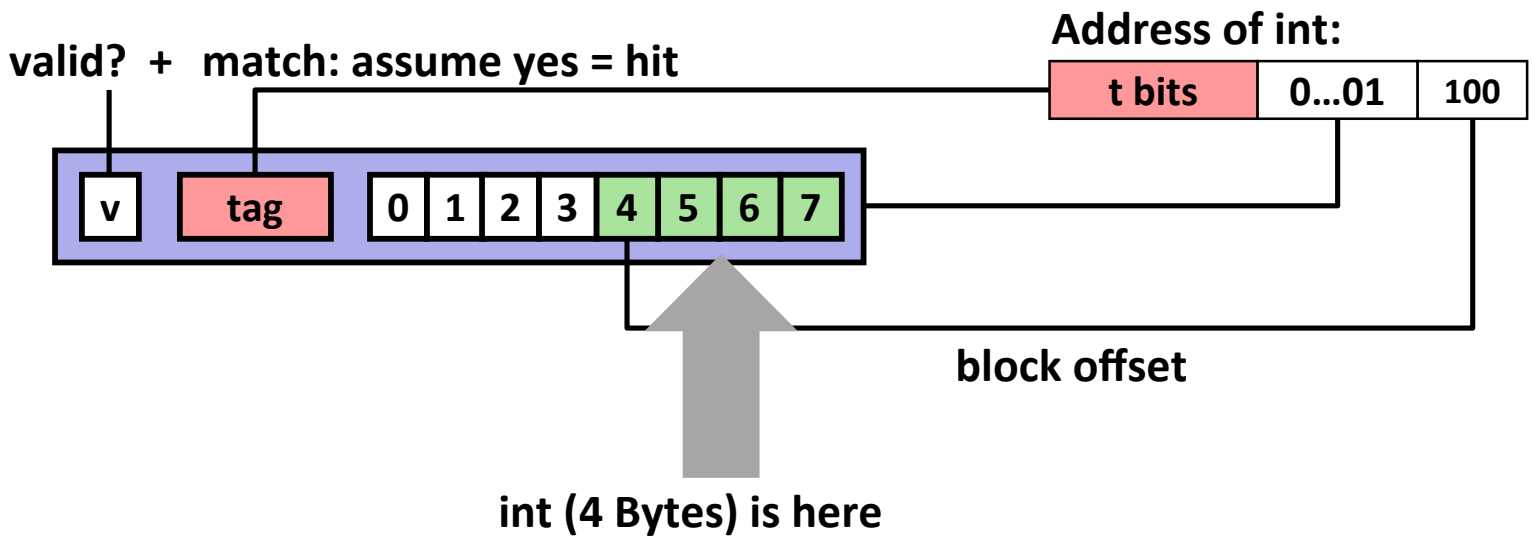
Example: direct mapped cache (E = 1)

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

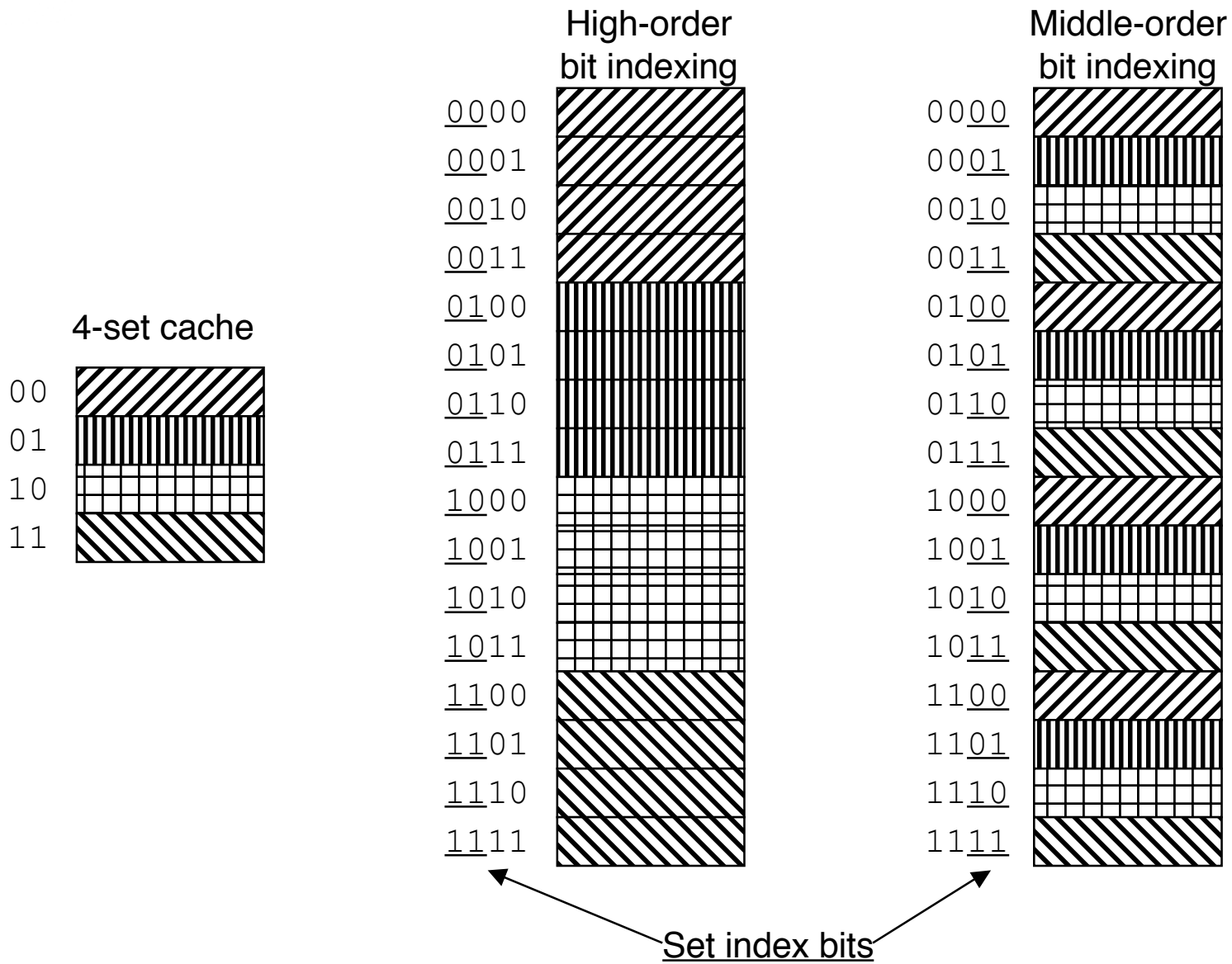
M=16 byte addresses, B=2 bytes/block,
S=4 sets, E=1 Blocks/set

Address trace (reads, one byte per read):

0	**[0000$_2$],**	miss
1	**[0001$_2$],**	hit
7	**[0111$_2$],**	miss
8	**[1000$_2$],**	miss
0	**[0000$_2$]**	miss

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

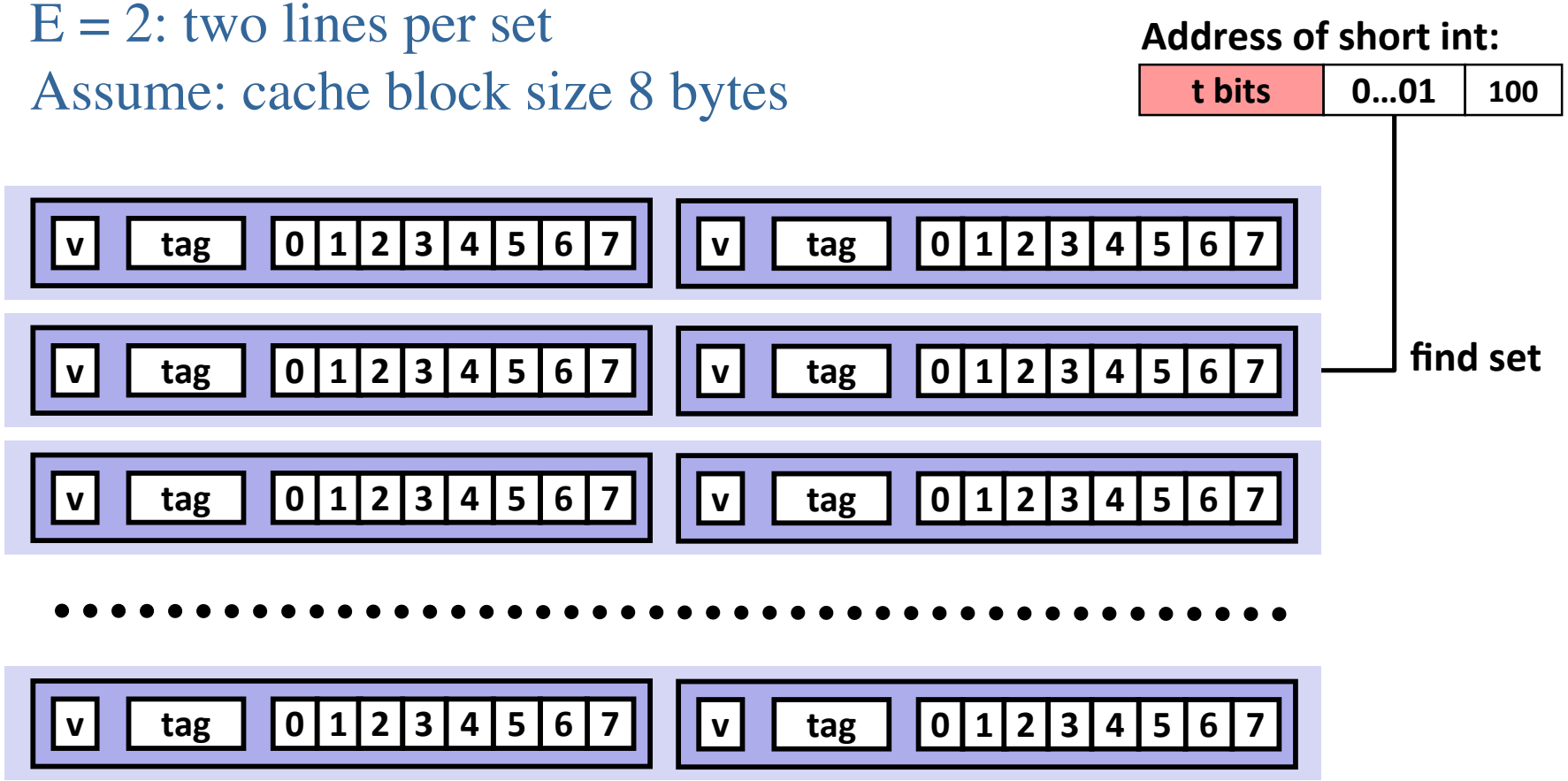
Why indexing with the middle bits?



E-way set associative cache

E = 2: two lines per set

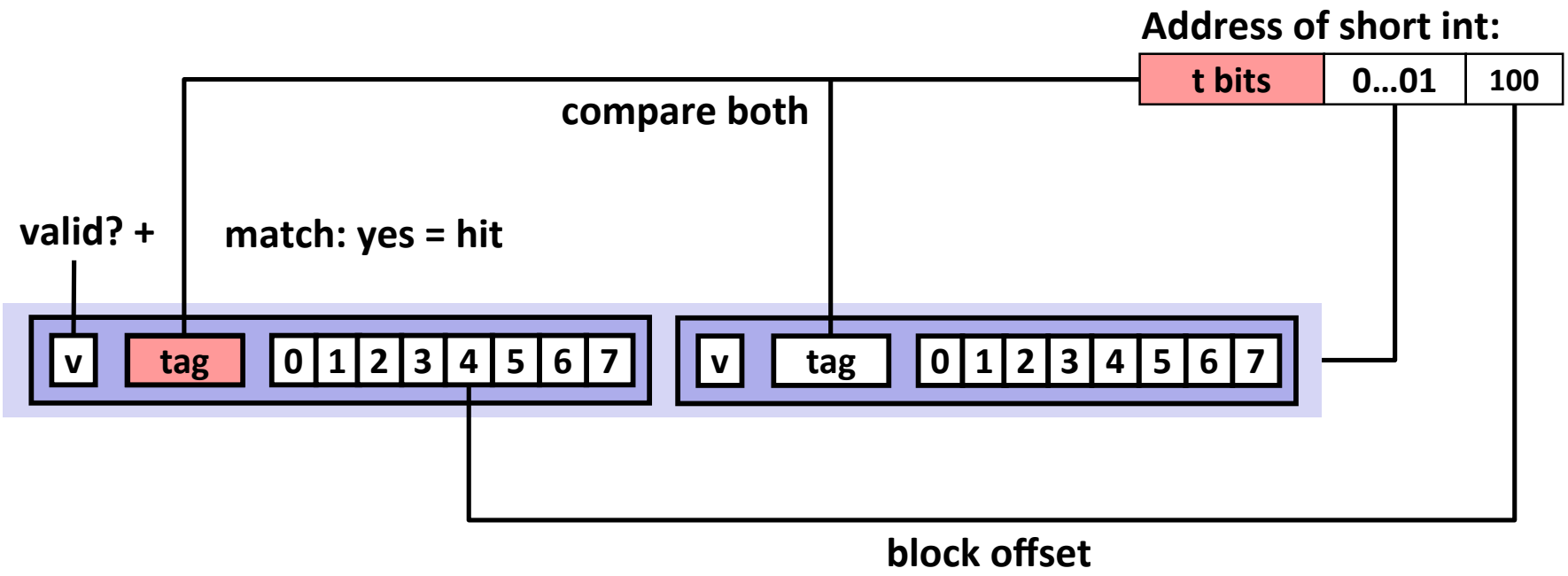
Assume: cache block size 8 bytes



E-way set associative cache

E = 2: two lines per set

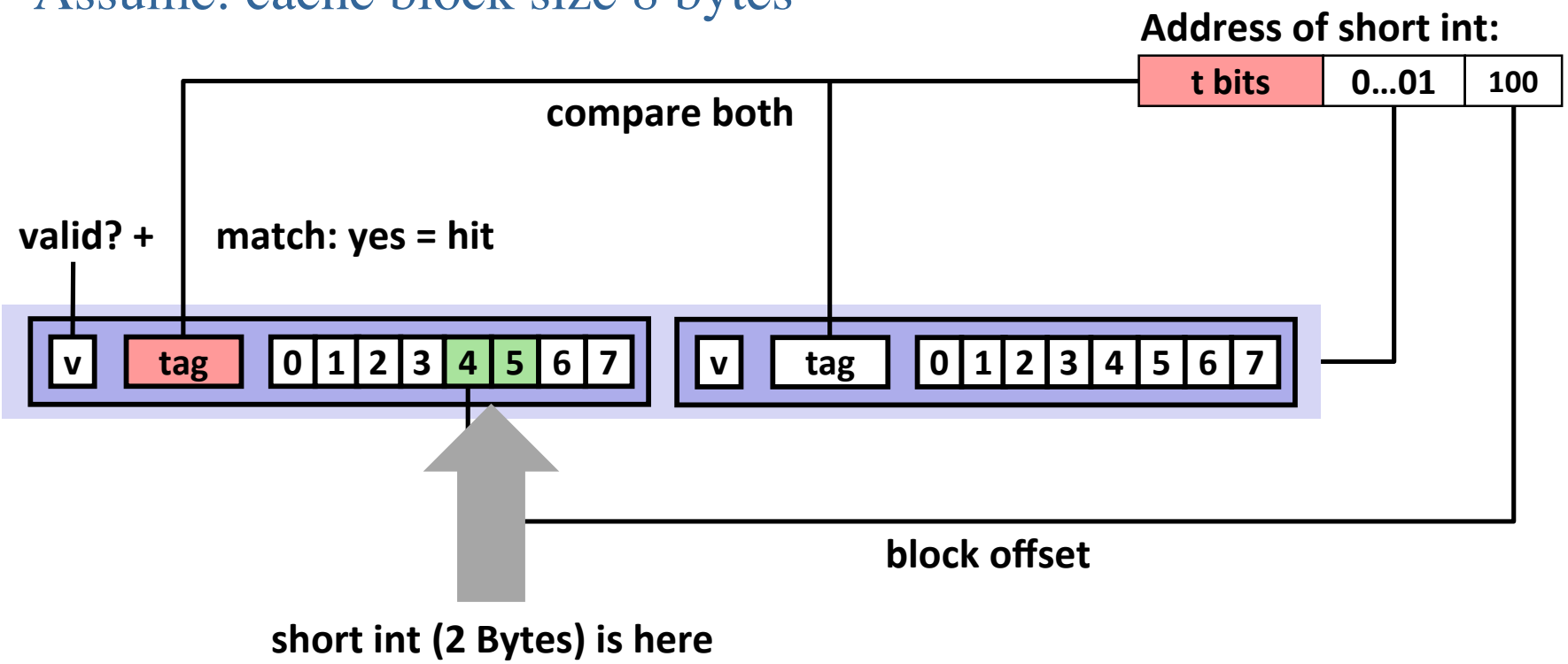
Assume: cache block size 8 bytes



E-way set associative cache

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		

Fully associative caches

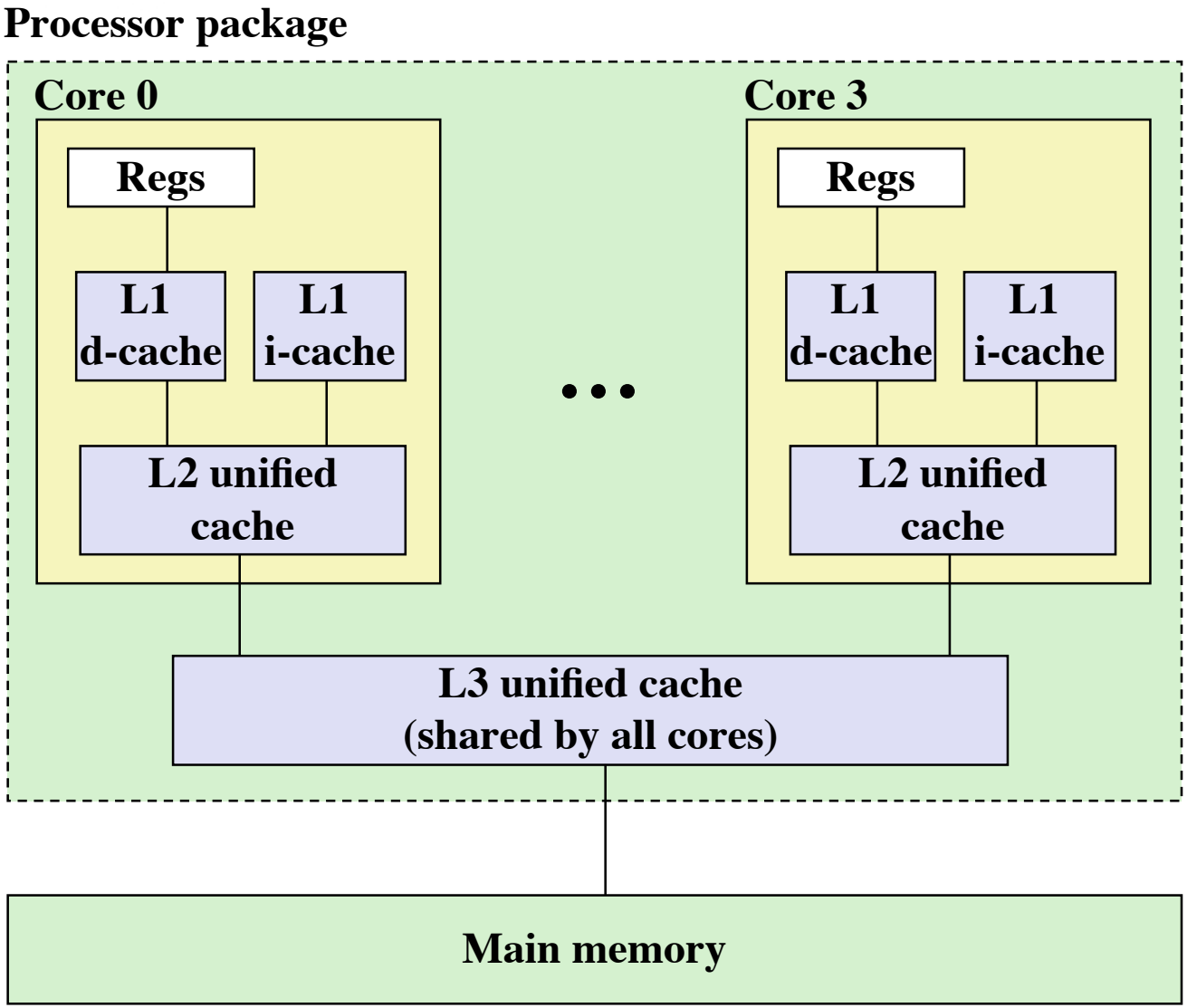
Recall C = S x E x B

S = 1 and thus E = C/B

All lines in the same set

Larger associativity implies longer access times, but less misses

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.

Kinds of cache misses

Cold (or compulsory) misses: happens when cache is empty, but not after cache has been warmed up

Conflict misses: cache is large enough, but addresses map to the same block
More frequent for smaller associativity

Capacity misses: working set is larger than cache size

Conflict misses

```
float dotprod(float x[8], float y[8])
{
    int i;
    float sum = 0.0;

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

    return sum;
}
```

Assume:

- S = 2,
- direct-mapped cache (E = 1)
- B = 16 bytes (4 floats)

	Address	Set index		Address	Set index
000000_2	0	0	100000_2	32	0
000100_2	4	0	100100_2	36	0
001000_2	8	0	101000_2	40	0
001100_2	12	0	101100_2	44	0
010000_2	16	1	110000_2	48	1
010100_2	20	1	110100_2	52	1
011000_2	24	1	111000_2	56	1
011100_2	28	1	111100_2	60	1

- Trashing!
- B bytes of padding at the end of each array
- OK if E>1

Credits

Computer systems: a programmer's perspective

Randal Bryant - David O'Hallaron

Pearson, second edition, 2010