

113: Architecture

Spring 2018

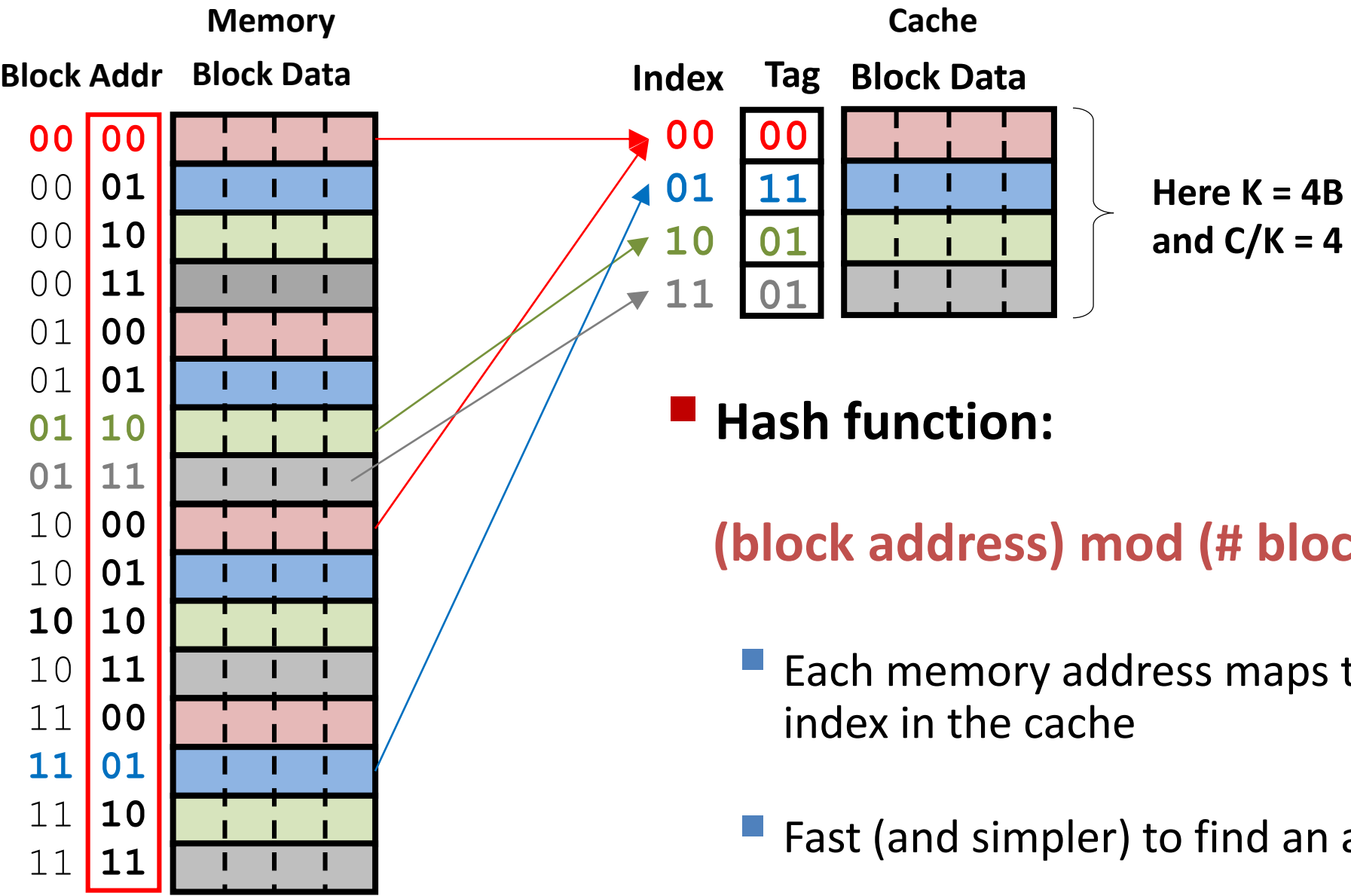
Lecture: Caches (Memory Hierarchy)

Instructor: Dr. Jana Giceva

Today: Caches and Memory Hierarchy

- Cache basics
- Principle of locality
- Memory hierarchies
- **Cache organization**
 - Direct-mapped (sets, index + tag)
 - Associativity (ways)
 - Replacement policy
 - Handling writes
- Program optimizations that consider caches

Direct-Mapped Cache

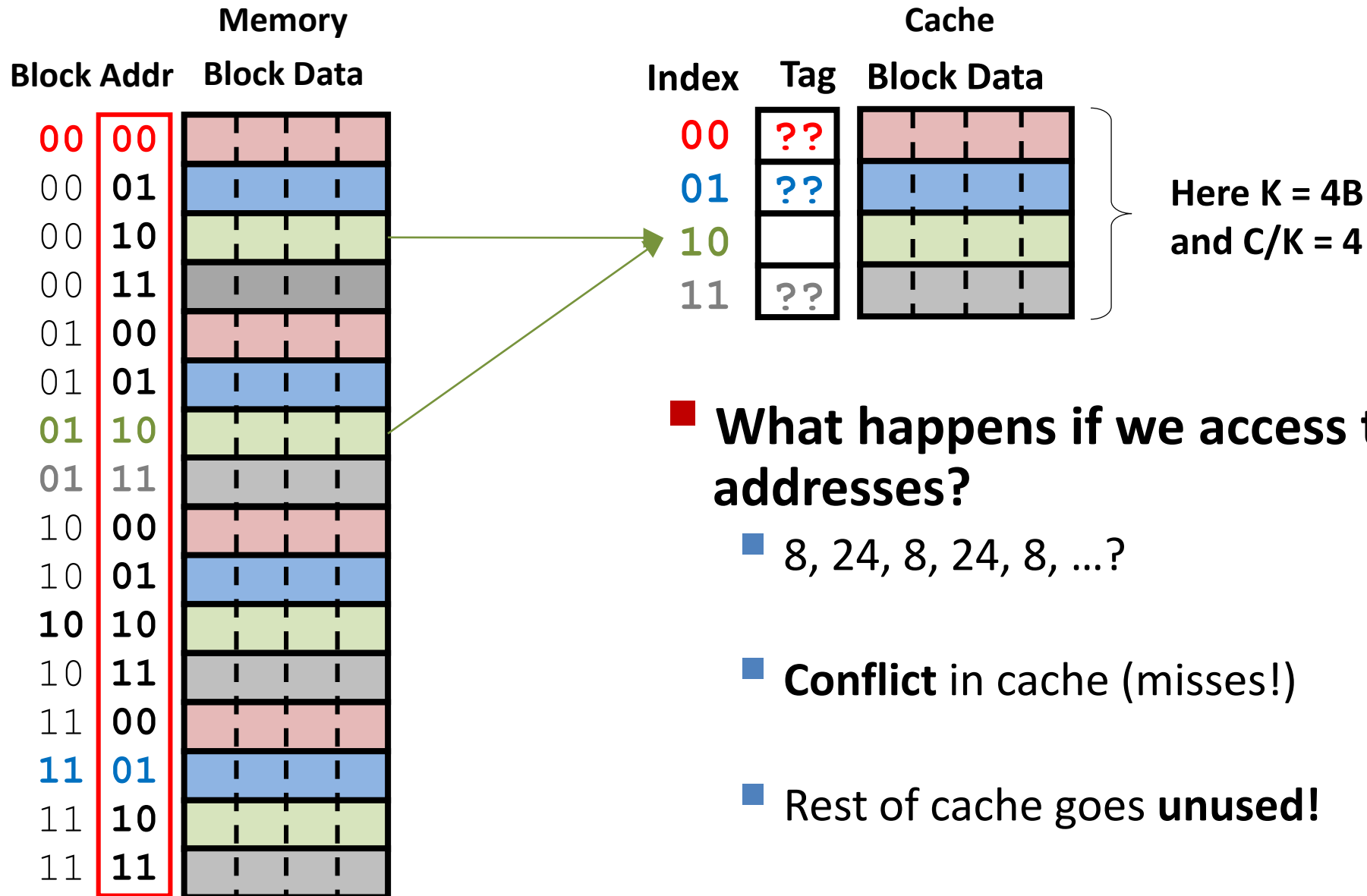


■ **Hash function:**

$(\text{block address}) \bmod (\# \text{ blocks in cache})$

- Each memory address maps to exactly one index in the cache
- Fast (and simpler) to find an address

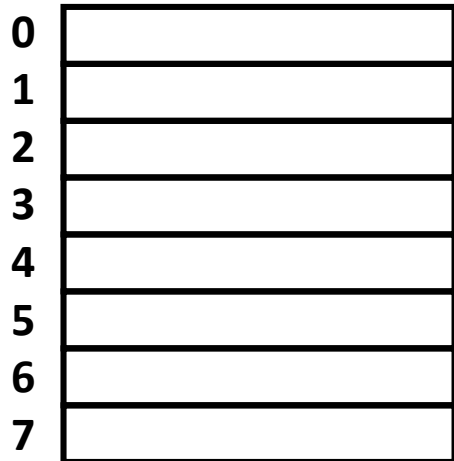
Direct-Mapped Cache Problem



Associativity

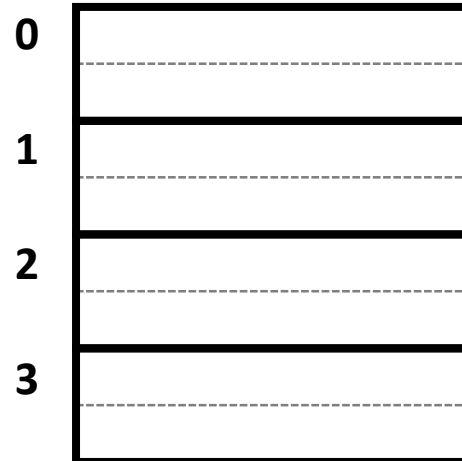
- What if we could store data in any place in the cache?
 - More complicated hardware = more power consumed and slower
- So we *combine* two ideas:
 - Each address maps to exactly one **set**
 - Each set can store block in more than one **way**

1 way, 8 sets
1 block each

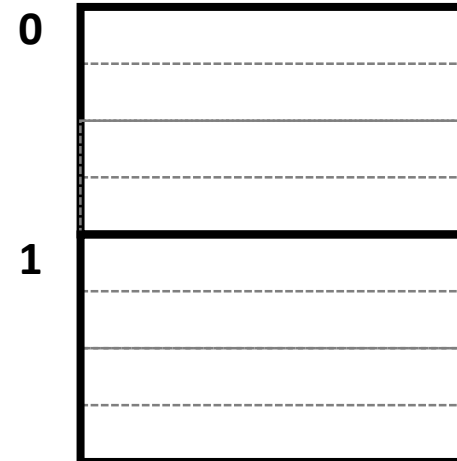


direct mapped

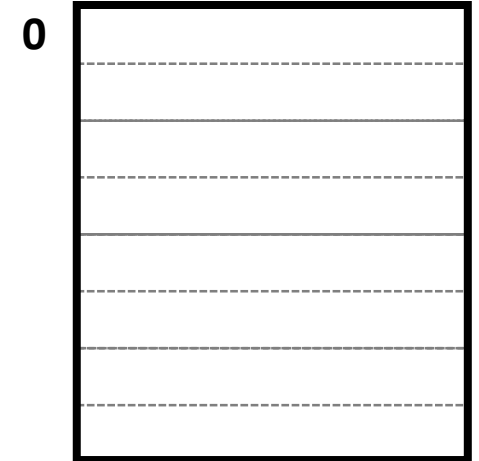
2 way, 4 sets
2 blocks each



4 way, 2 sets
4 blocks each



8 way, 1 sets
8 blocks

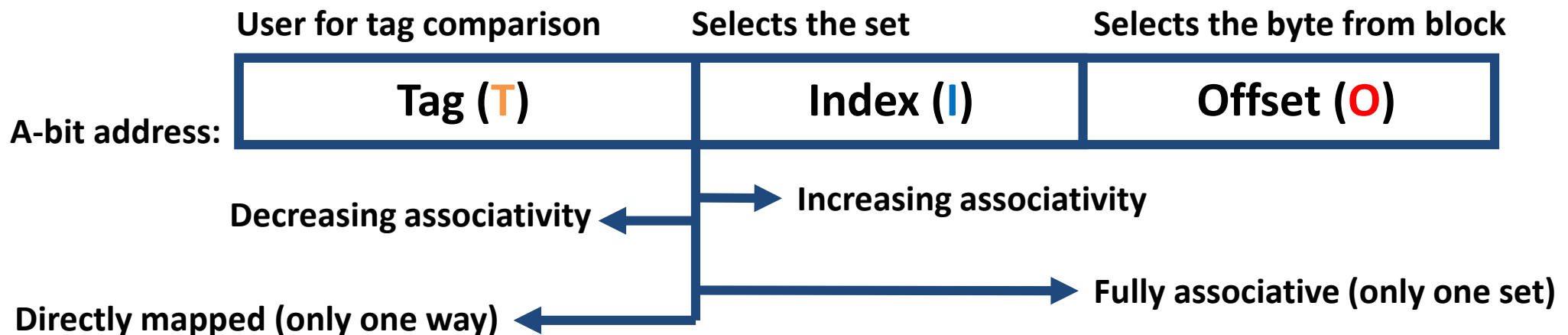


fully associative

Cache Organization (3)

Cache notation:
C – size of cache
K – block size
N – associativity

- **Associativity (N): number of ways for each set**
 - Such a cache is called an “N-way set associative cache”
 - We now index into cache sets, of which there are $C/K/N$
 - Use lowest $\log_2(C/K/N) = I$ bits of block address
 - Direct-mapped: $N=1$, so $I = \log_2(C/K)$ as we saw previously
 - Fully-associative: $N=C/K$, so $I = 0$ bits

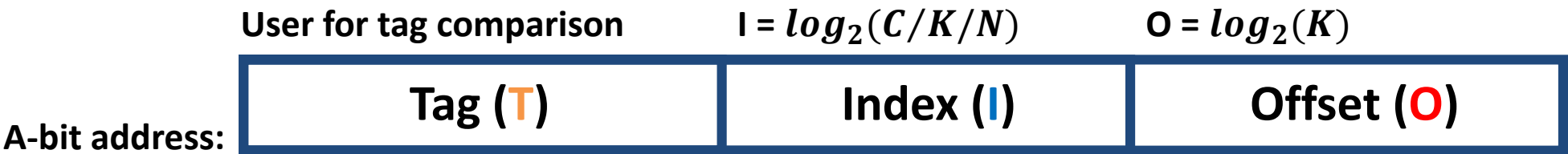


Example Placement

Block size (K)	16 B
Capacity (C/K)	8 blocks
Address (A)	16 bits

Where would data from address 0x1833 be placed?

Binary: 0b 0001 1000 0011 0011



Direct-mapped

Set	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

2-way set associative

Set	Tag	Data
0		
1		
2		
3		

4-way set associative

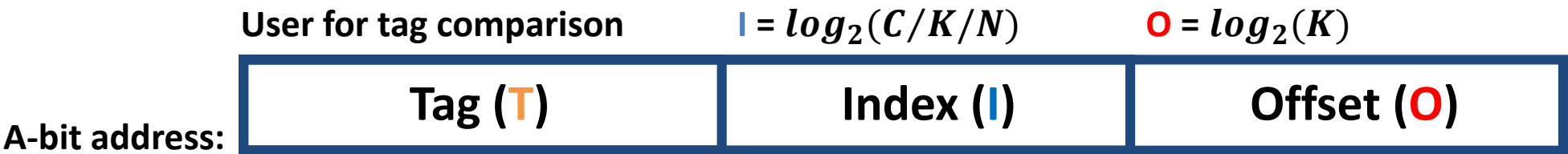
Set	Tag	Data
0		
1		

Example Placement

Block size (K)	16 B
Capacity (C/K)	8 blocks
Address (A)	16 bits

Where would data from address 0x1833 be placed?

Binary: 0b 0001 1000 0011 0011



$I = 3$

Direct-mapped

Set	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

$I = 2$

2-way set associative

Set	Tag	Data
0		
1		
2		
3		

$I = 1$

4-way set associative

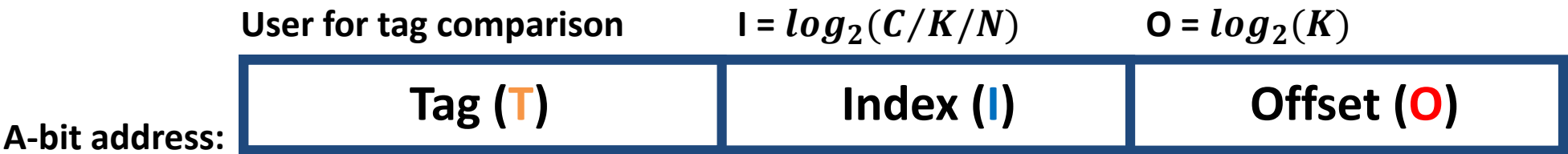
Set	Tag	Data
0		
1		

Example Placement

Block size	16 B
Capacity	8 blocks
Address	16 bits

Where would data from address 0x1833 be placed?

Binary: 0b 0001 1000 0011 0011



Direct-mapped

Set	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

2-way set associative

Set	Tag	Data
0		
1		
2		
3		

4-way set associative

Set	Tag	Data
0		
1		

Block Replacement

- Any *empty* block in the correct set may be used to store block
- If there are no empty blocks, which one should we replace?
 - No choice for direct-mapped caches
 - Caches typically use something close to *least recently used (LRU)* (hardware usually implements “*not most recently used*”)

Direct-mapped

Set	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

2-way set associative

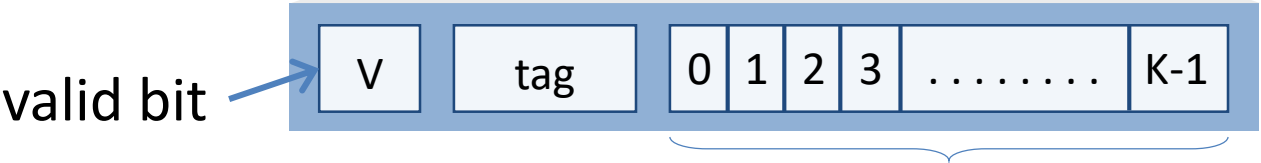
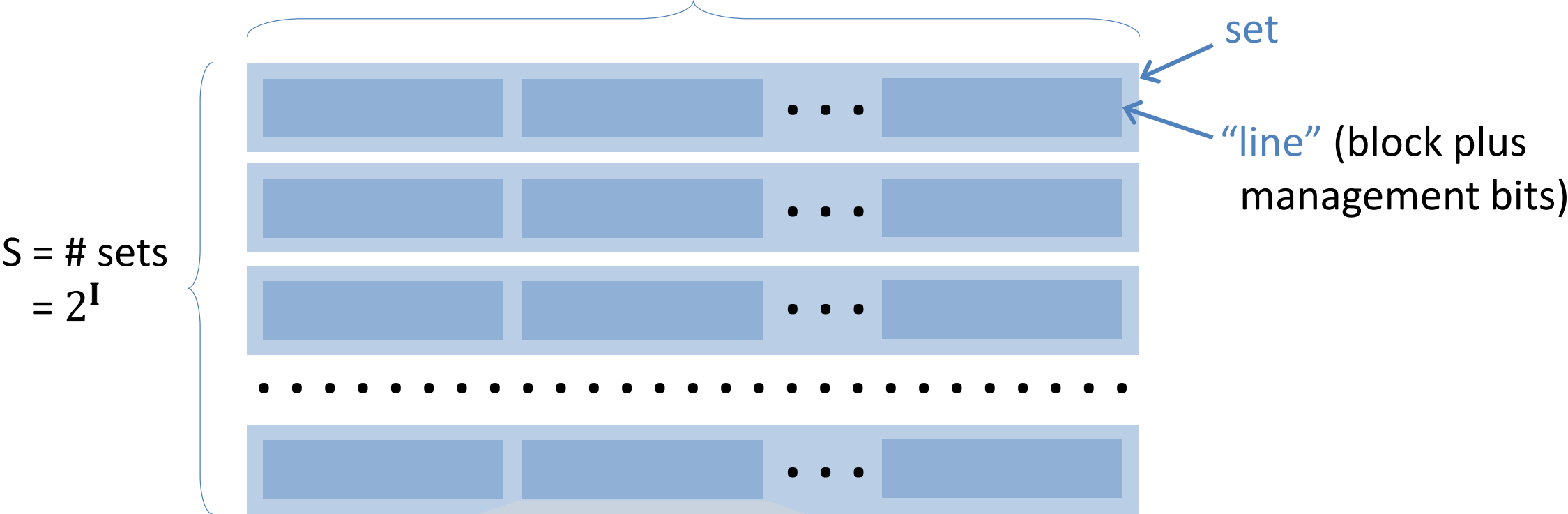
Set	Tag	Data
0		
1		
2		
3		

4-way set associative

Set	Tag	Data
0		
1		

General Cache Organisation (S, N, K)

N = blocks/lines per set

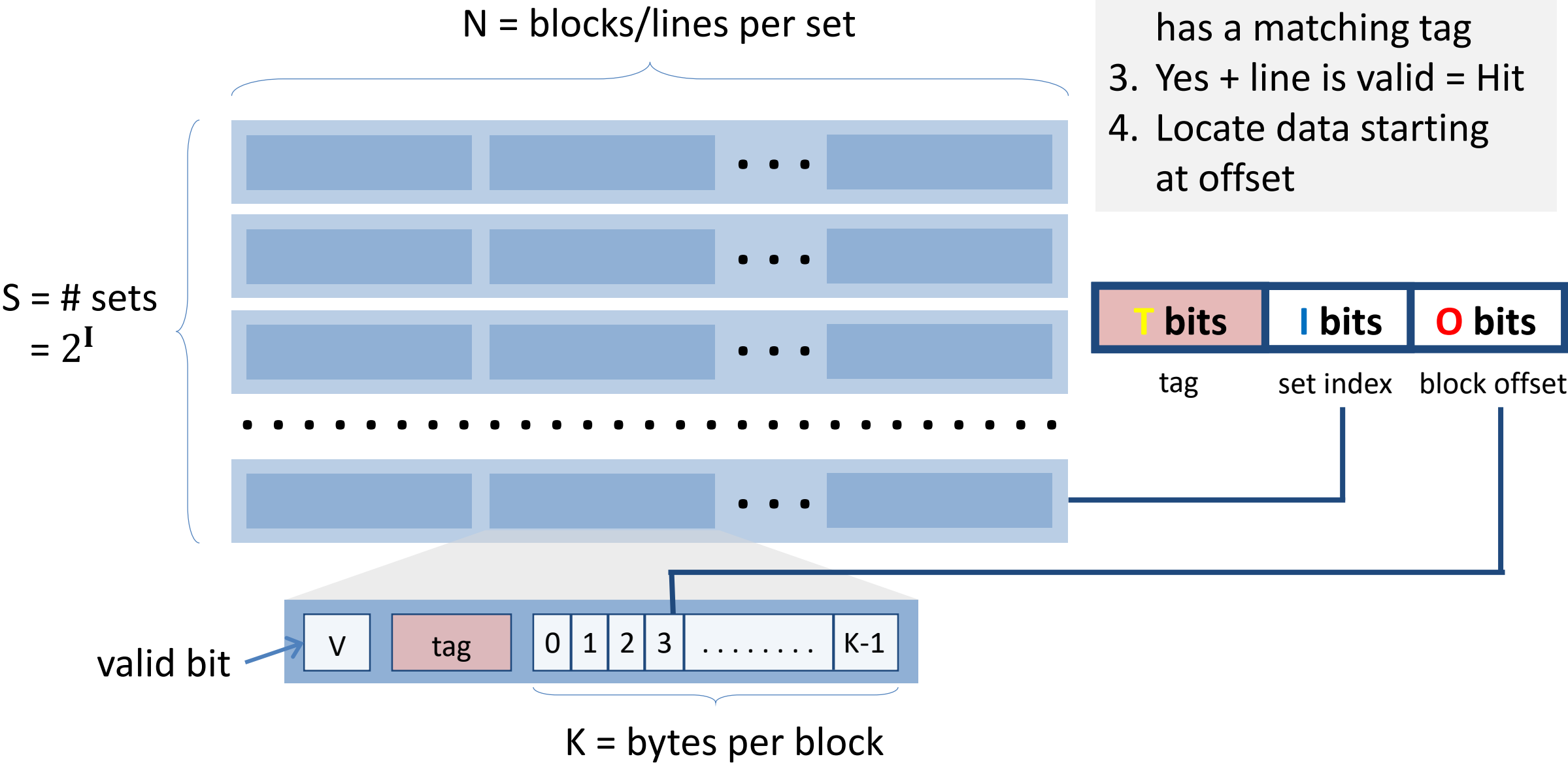


K = bytes per block

Cache size:

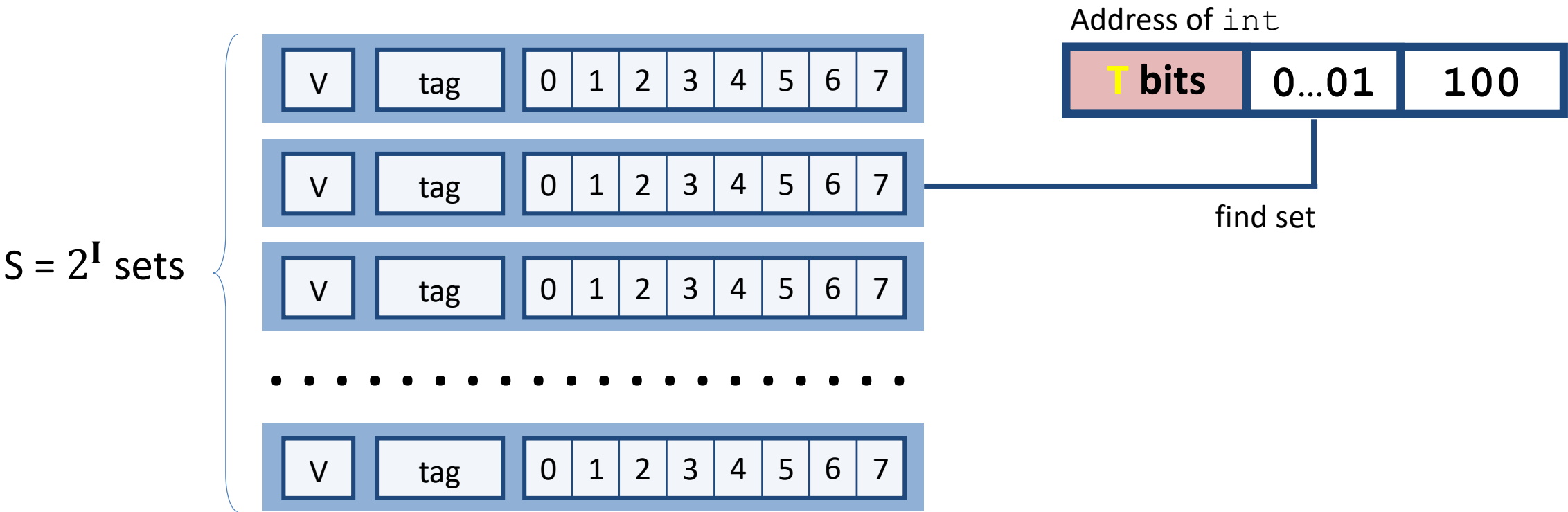
$C = S \times N \times K$ data bytes
(doesn't include V or Tag)

Cache Read



Example: Direct-Mapped Cache (N=1)

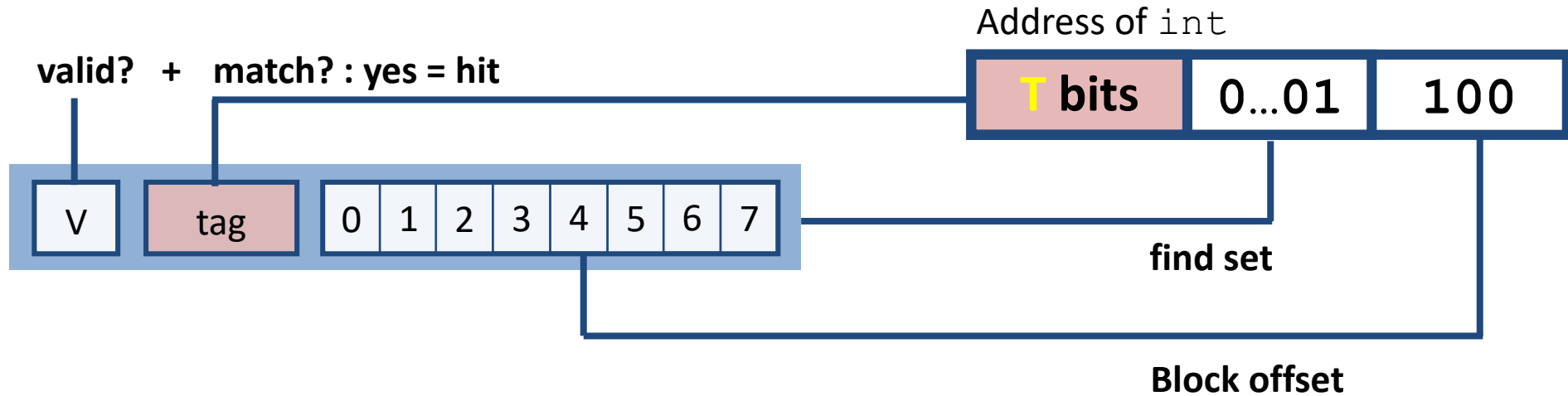
Direct-mapped: One line/block per set
Block size K = 8 bytes



Example: Direct-Mapped Cache (N=1)

Direct-mapped: One line/block per set

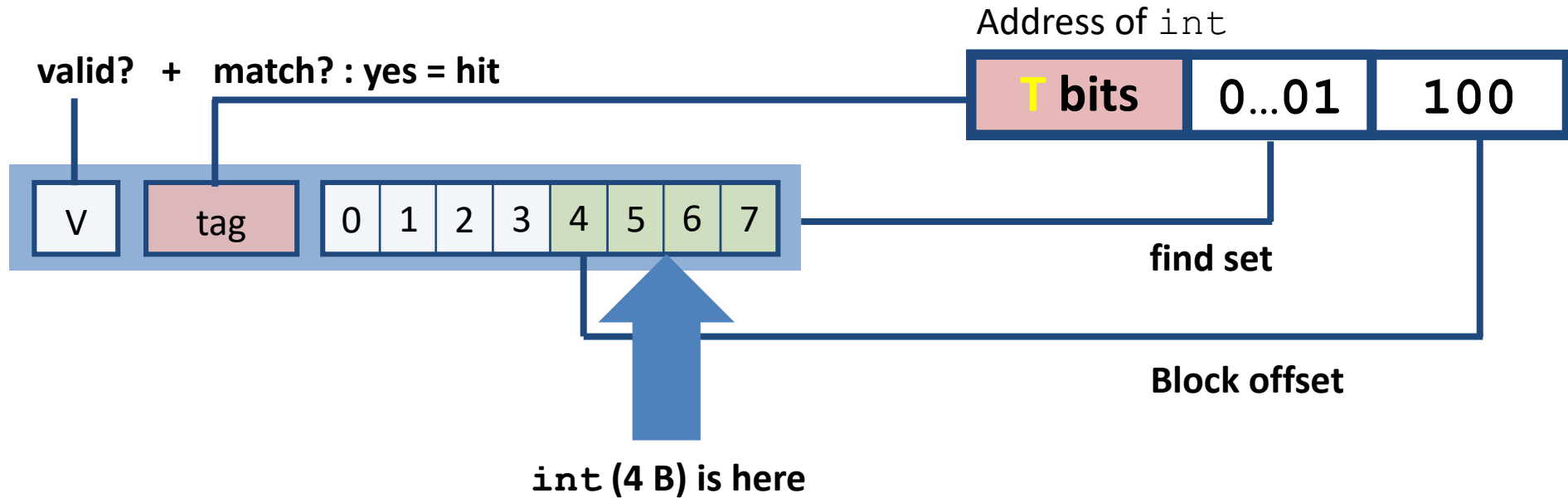
Block size $K = 8$ bytes



Example: Direct-Mapped Cache (N=1)

Direct-mapped: One line/block per set

Block size K = 8 bytes

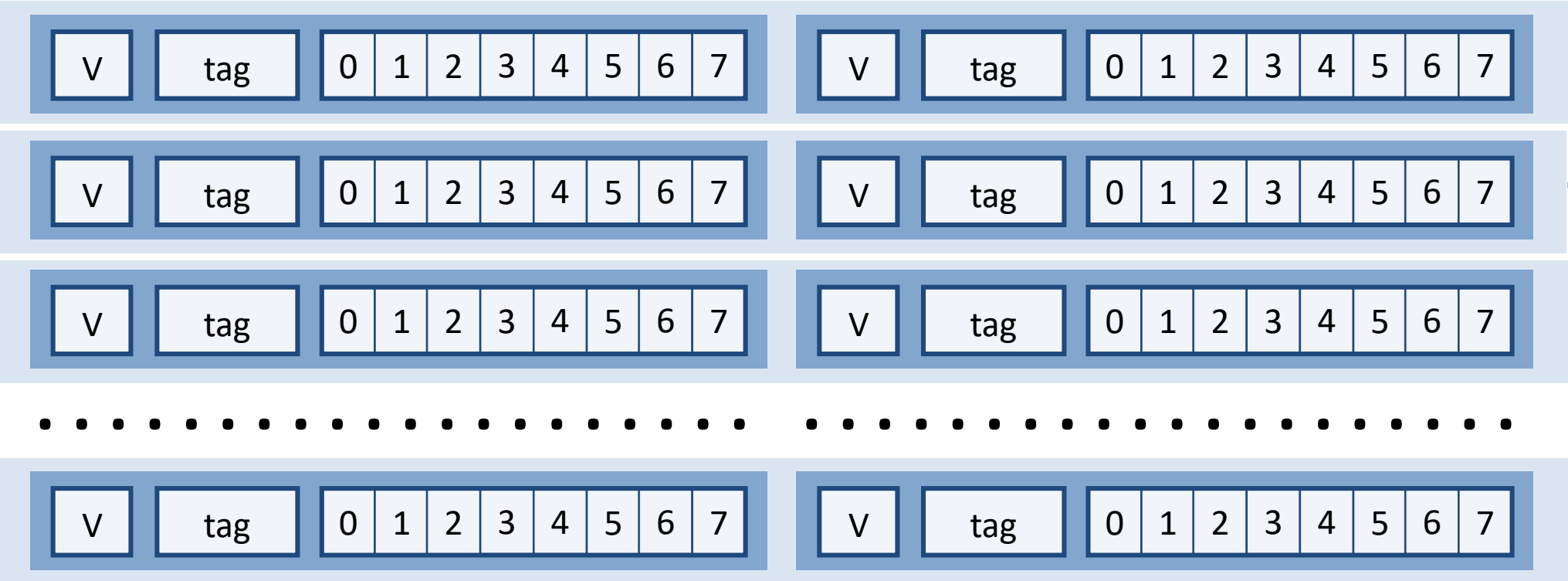


No match?

- Then old line gets evicted and replaced

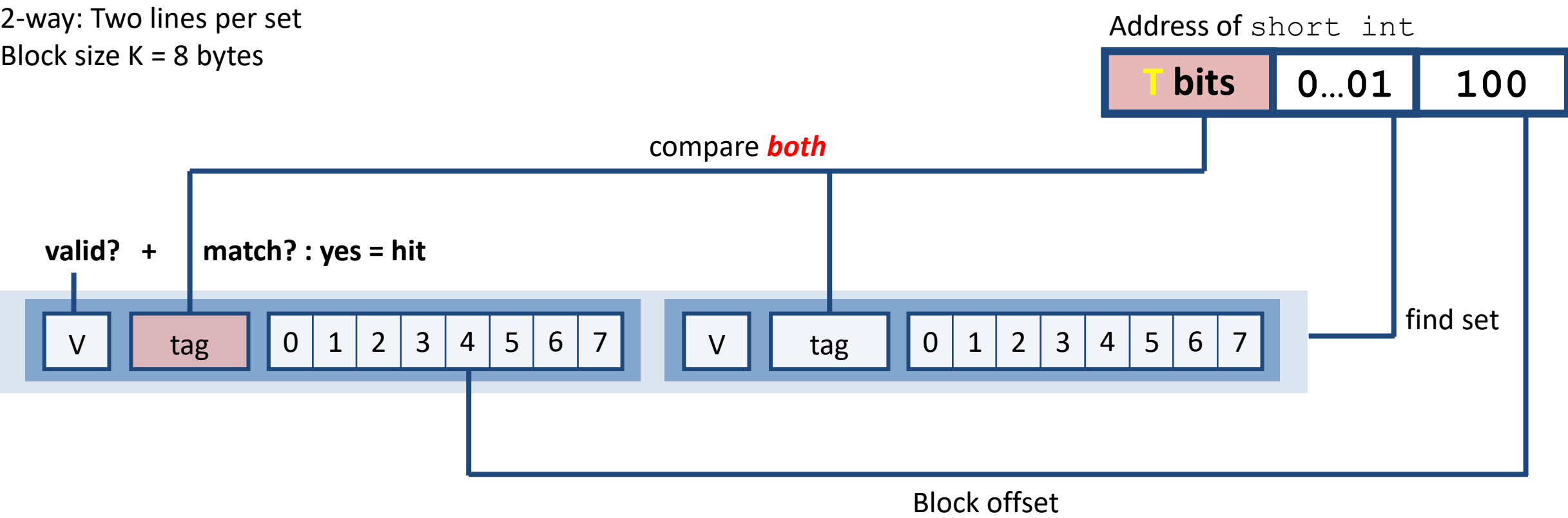
Example: Set-Associative Cache (N=2)

2-way: Two lines per set
Block size K = 8 bytes



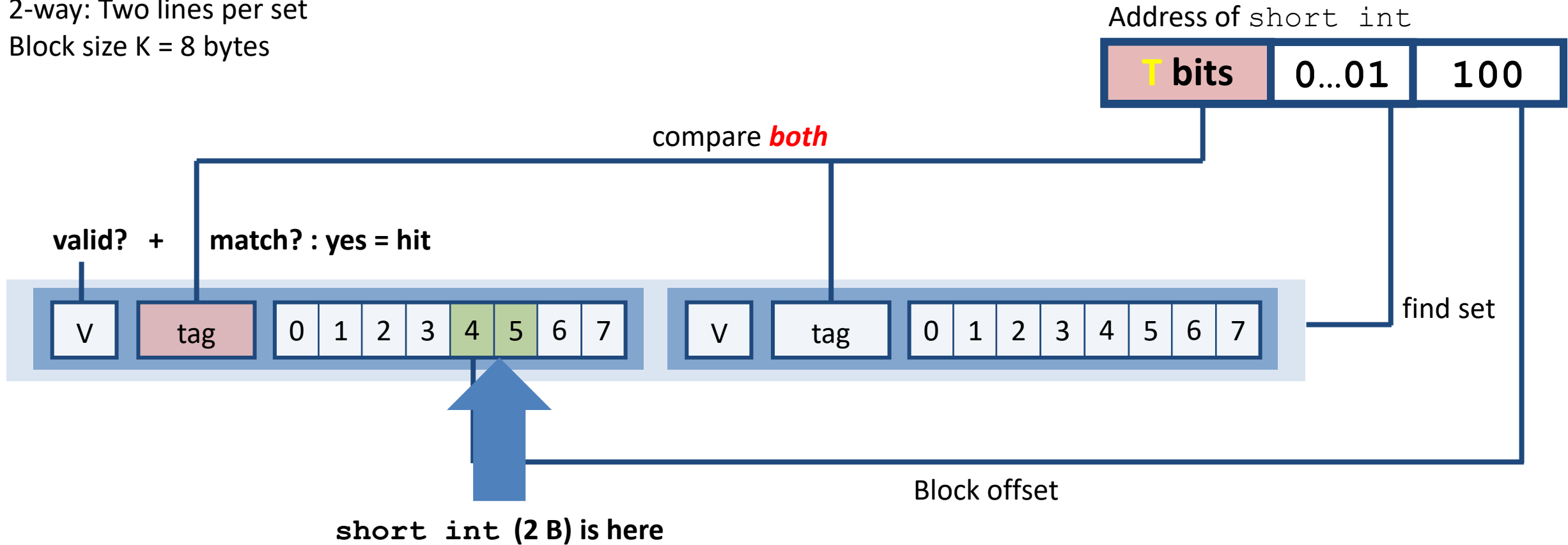
Example: Set-Associative Cache (N=2)

2-way: Two lines per set
Block size K = 8 bytes



Example: Set-Associative Cache (N=2)

2-way: Two lines per set
Block size K = 8 bytes



No match?

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

Types of Cache Misses: 3 C's!

- **Compulsory** (cold) miss:

- Occurs on first access to a block

- **Conflict** miss:

- Conflict misses occur when the cache is large enough, but multiple data objects all map to the same slot.
 - E.g., referencing blocks 0, 8, 0, 8, ... could miss every time
- Direct-mapped caches have more conflict misses than N-way set-associative

- **Capacity** miss:

- Occurs when the set of active cache blocks (the **working set**) is larger than the cache
- Note: Fully-associative only has Compulsory and Capacity misses

What to do on a write hit?

- Multiple copies of data exist. What is the problem with that?

- **Write-through**

- Write immediately to memory and all caches in between
- Memory is always consistent with the cache copy
- Slow: what if the same value (or line!) is written several times

- **Write-back**

- Defer write to memory until line is evicted (replaced)
- Need a dirty bit
 - Indicates line is different from memory
- Higher performance (but more complex)

What to do on a write-miss?

- **Write-allocate** (load into cache, update line in cache)
 - Good if more writes to the location follow
 - More complex to implement
 - May evict an existing value
 - Common with write-back caches
- **No-write-allocate** (writes immediately to memory)
 - Simpler to implement
 - Slower code (bad if value consistently re-read)
 - Seen with write-through caches

Real caches: Intel Core i7-5960X (Haswell)

- All caches have a block/line size of 64 bytes

- L1 i-cache and d-cache:

- 32 KiB, 8-way set-associative
- i-cache: no writes, d-cache: write-back
- Access: 4 cycles

- L2 unified cache:

- 256 KiB, 8-way set-associative
- private, write-back
- Access: 11 cycles

- L3 unified cache: (shared among multiple cores)

- 8 MiB, 16-way set-associative
- shared, write-back
- Access: 30-40 cycles



Slower, but more likely to hit

Software caches are more flexible

■ Examples:

- file system buffer caches, web browser caches, etc.
- Content-delivery networks (CDN): cache for the internet (e.g., Netflix)

■ 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

Today: Caches and Memory Hierarchy

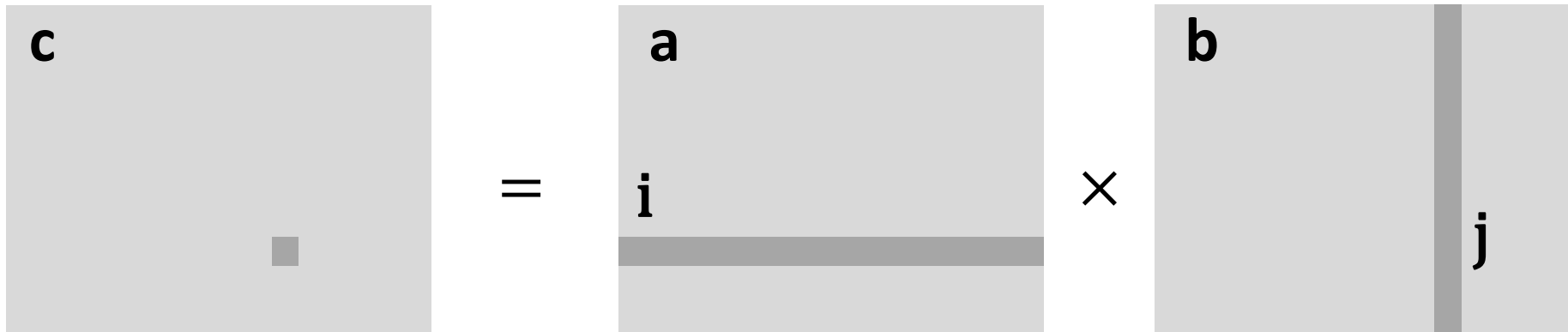
- Cache basics
- Principle of locality
- Memory hierarchies
- Cache organization
 - Direct-mapped (sets, index + tag)
 - Associativity (ways)
 - Replacement policy
 - Handling writes
- **Program optimizations that consider caches**

Optimizations for the memory hierarchy

- **Write code that has locality**
 - *Spatial*: access data contiguously
 - *Temporal*: make sure access to the same data is not too far apart in time
- **How to achieve this?**
 - Adjust memory access in *code* (software) to improve miss rate (MR)
 - Requires knowledge of *both* how caches work as well as your system's parameters
 - Proper choice of algorithm
 - Loop transformations

Example: matrix multiplication

```
/* 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++) // move along rows of a  
        for (j = 0; j < n; j++) // move along columns of b  
            for (k = 0; k < n, k++)  
                c[i*n + j] += a[i*n + k] * b[k*n + j];  
}
```



Cache miss analysis

1. Read from $a(i, :)$
 2. Read from $b(:, j)$
- Ignoring matrix c for now

Assume:

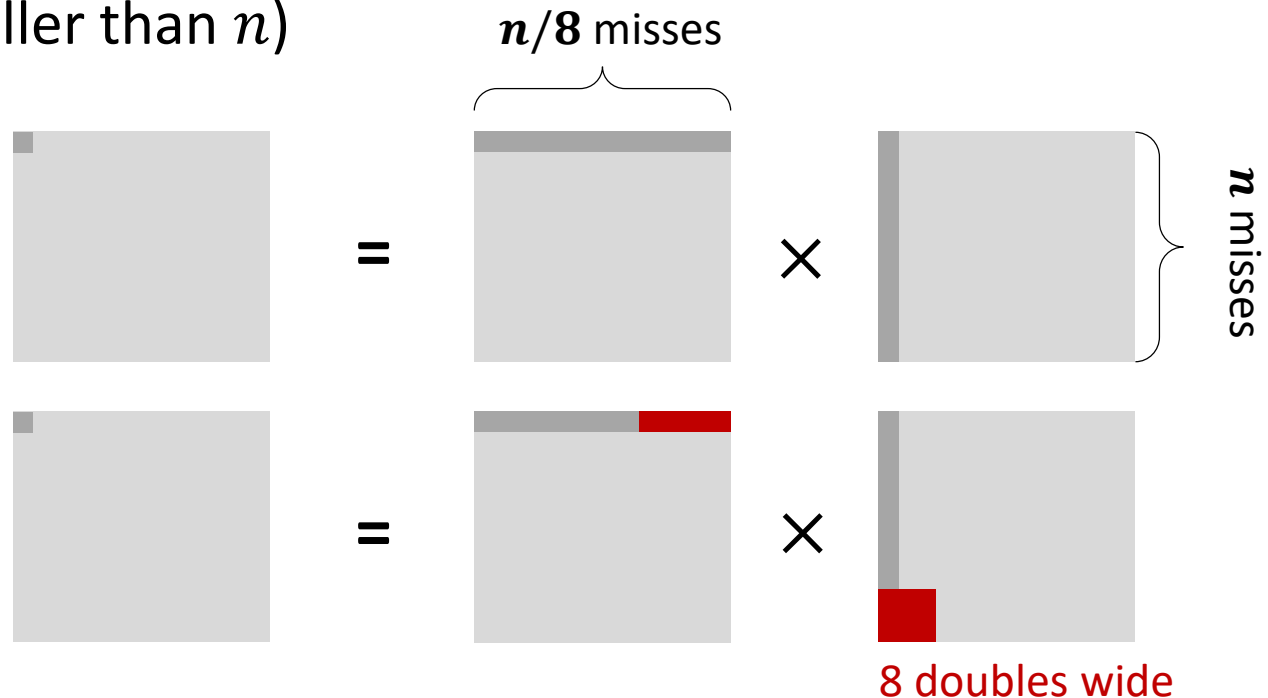
- Square matrix ($n \times n$), elements are doubles
- Cache block size $K=64$, 8 doubles in a block
- Cache size $C \ll n$ (much smaller than n)

First iteration:

- $\frac{n}{8} + n = \frac{9n}{8}$ misses

- Afterwards **in cache**:
(schematic)

- Total misses: $\frac{9n}{8} \times n^2 = \frac{9}{8}n^3$



Linear Algebra to the Rescue (1)

- Can get the same result of matrix multiplication by splitting the matrices into smaller submatrices (matrix “blocks”)
- For example, multiply two 4×4 matrices:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \text{ with } B \text{ defined similarly.}$$

$$AB = \begin{bmatrix} (A_{11}B_{11} + A_{12}B_{21}) & (A_{11}B_{12} + A_{12}B_{22}) \\ (A_{21}B_{11} + A_{22}B_{21}) & (A_{21}B_{12} + A_{22}B_{22}) \end{bmatrix}.$$

Linear Algebra to the Rescue (2)

C_{11}	C_{12}	C_{13}	C_{14}
C_{21}	C_{22}	C_{23}	C_{24}
C_{31}	C_{32}	C_{33}	C_{34}
C_{41}	C_{42}	C_{43}	C_{44}

A_{11}	A_{12}	A_{13}	A_{14}
A_{21}	A_{22}	A_{23}	A_{24}
A_{31}	A_{32}	A_{33}	A_{34}
A_{41}	A_{42}	A_{43}	A_{44}

B_{11}	B_{12}	B_{13}	B_{14}
B_{21}	B_{22}	B_{23}	B_{24}
B_{31}	B_{32}	B_{33}	B_{34}
B_{41}	B_{42}	B_{43}	B_{44}

- **Matrices of size $n \times n$, split into 4 blocks of size r ($n = 4r$)**

$$C_{22} = A_{21}B_{12} + A_{22}B_{22} + A_{23}B_{32} + A_{24}B_{42} = \sum_k A_{2k} \times B_{k2}$$

- **Multiplication operates on small “block” matrices**

- Choose size so that they fit in the cache
- This technique called “*cache blocking*”

Blocked Matrix Multiply

```
/* move by rxr BLOCKS now */
for (i = 0; i < n; i+=r)
    for (j = 0; j < n; j+=r)
        for (k = 0; k < n, k+=r)
            /* block matrix multiplication */
            for (ib = i; ib < i+r; ib++)
                for (jb = j; jb < j+r; jb++)
                    for (kb = k; kb < k+r; kb++)
                        c[ib*n + jb] += a[ib*n + kb] * b[kb*n + jb]
```

- **Blocked version of the naïve algorithm**
 - r = block matrix size (assume r divides n evenly)
- **6 nested loops may seem less efficient, but leads to a much faster code!!**

Cache Miss Analysis (Blocked)

■ Assume:

- Square matrix ($n \times n$), elements are doubles
- Cache block size $K = 64$, 8 doubles in a cache block
- Cache size $C \ll n$ (much smaller than n)
- Three blocks ($r \times r$) fit into cache: $3r^2 < C$

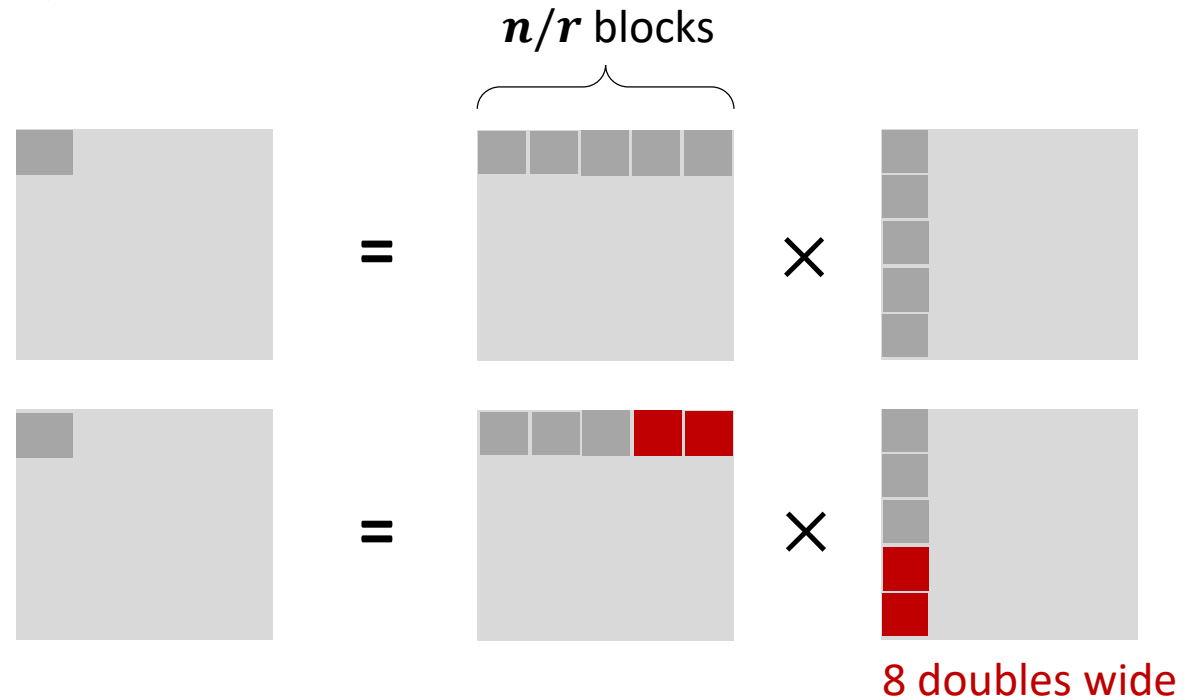
r^2 elements per block, 8 blocks in cache

■ First (block) iteration:

- $\frac{r^2}{8}$ misses for each block
- $\frac{2n}{r} \times \frac{r^2}{8} = \frac{nr}{4}$ (again omitting matrix c)

n/r blocks in row and column

- Afterwards **in cache** (schematic):



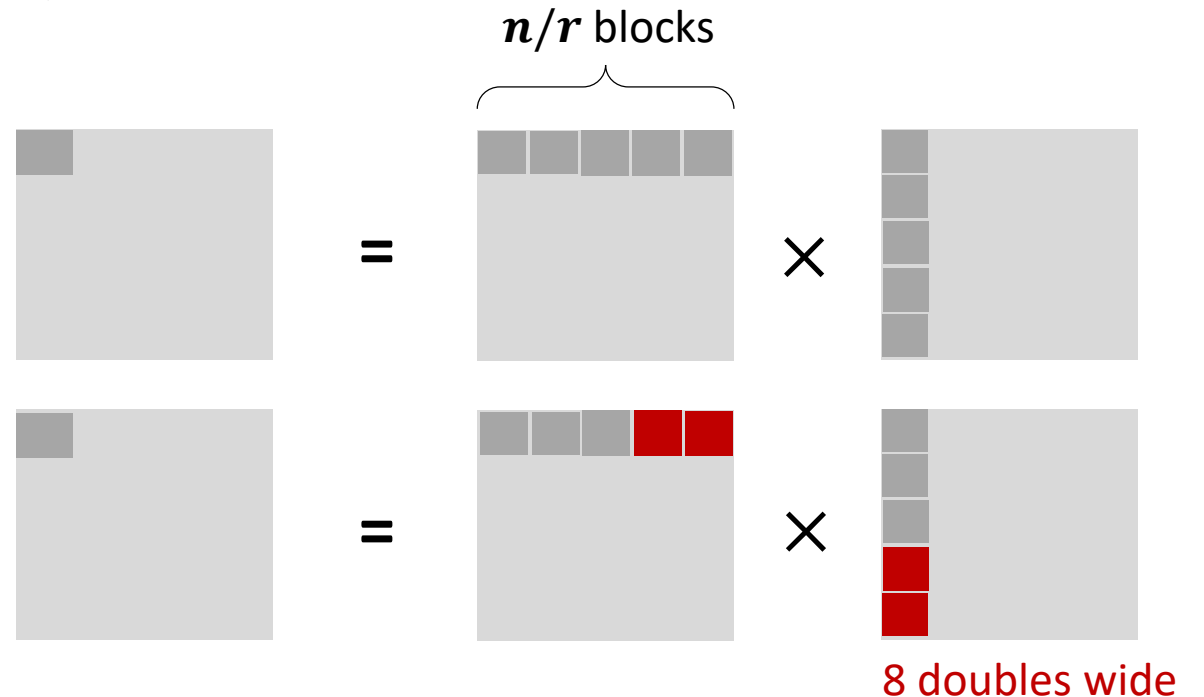
Cache Miss Analysis (Blocked)

■ Assume:

- Square matrix ($n \times n$), elements are doubles
- Cache block size $K = 64$, 8 doubles in a cache block
- Cache size $C \ll n$ (much smaller than n)
- Three blocks ($r \times r$) fit into cache: $3r^2 < C$

■ First (block) iteration:

- $\frac{r^2}{8}$ misses for each block
- $\frac{2n}{r} \times \frac{r^2}{8} = \frac{nr}{4}$ (again omitting matrix c)
- Total misses:
$$\frac{nr}{4} \times \left(\frac{n}{r}\right)^2 = \frac{n^3}{(4r)}.$$



Matrix Multiply Summary

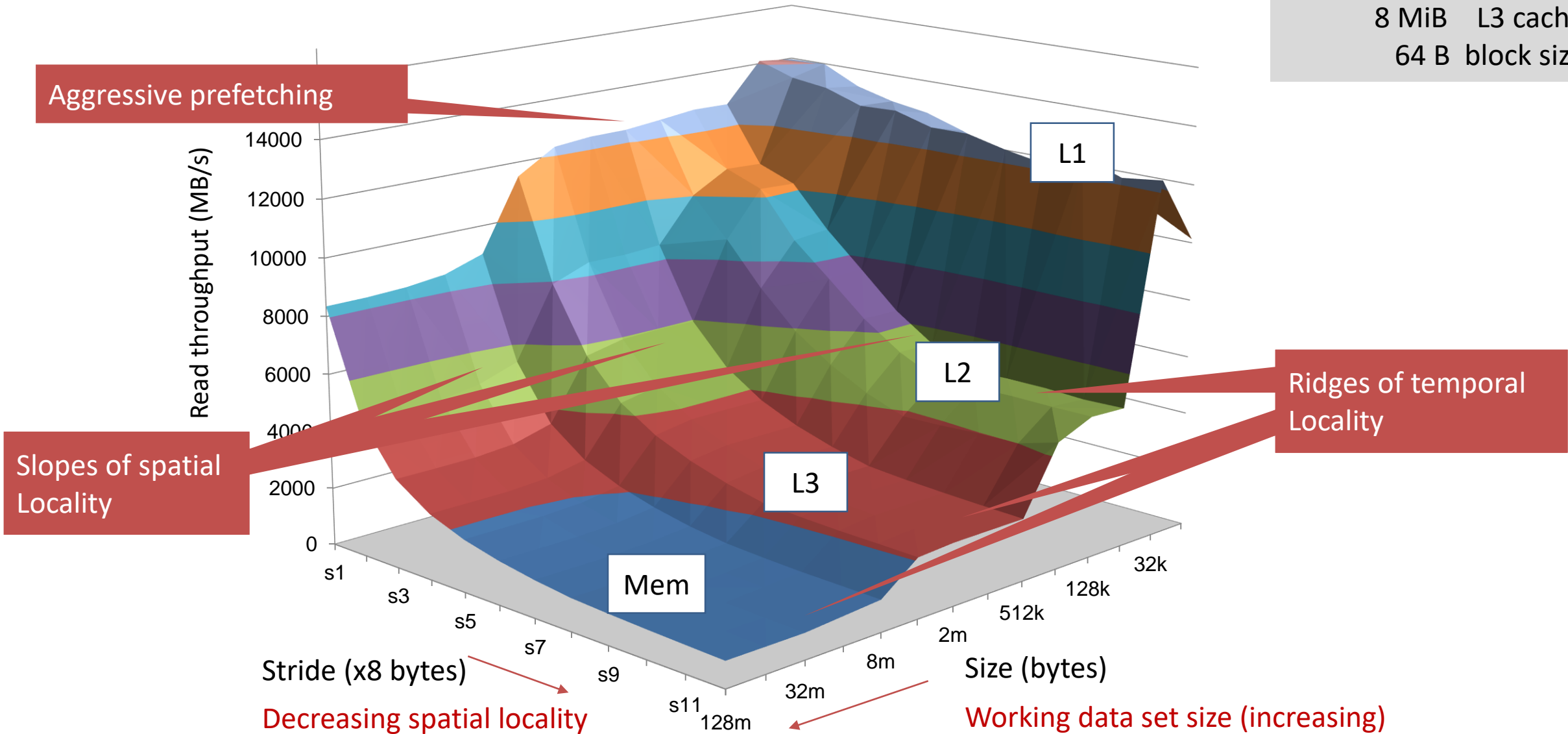
- **Naïve:** $(9/8) \times n^3$
- **Blocked:** $1/(4r) \times n^3$
 - If $r = 8$, difference is $4 * 8 * \frac{9}{8} = 36x$
 - If $r = 16$, difference is $4 * 16 * \frac{9}{8} = 72x$
- **Blocking optimization only works if the blocks fit in the cache**
 - Suggests larger possible block size up to limit $3r^2 \leq C$
- **Matrix multiplication has inherent temporal locality:**
 - Input data: $3n^2$, computation $2n^3$
 - Every array element used $O(n)$ times!
 - But program has to be written properly

Cache-Friendly Code

- **Programmer can optimise for cache performance**
 - How data structures are organised
 - How data are accessed:
 - Nested loop structure
 - Blocking is a general technique
- **All systems favour “cache-friendly code”**
 - Getting absolute optimum performance is very platform specific
 - Cache sizes, cache block size, associativity, etc.
 - Can get most of the advantage with generic code:
 - Keep working set reasonably small (temporal locality)
 - Use small strides (spatial locality)
 - Focus on inner loop cycle

The Memory Mountain

Core i7 Haswell 2.1 GHz
32 KiB L1 cache
256 KiB L2 cache
8 MiB L3 cache
64 B block size



Learn About Your Machine

■ Linux:

- **lscpu**
- `ls /sys/devices/system/cpu/cpu0/cache/index0/`
 - Ex: `cat /sys/devices/system/cpu/cpu0/cache/index*/size`
- `cat /proc/cpuinfo | grep cache | sort | uniq`

■ Windows:

- `wmic memcache get <query> (all values in KB)`
- Ex: `wmic memcache get MaxCacheSize`

■ Modern processor specs: <http://www.7-cpu.com/>