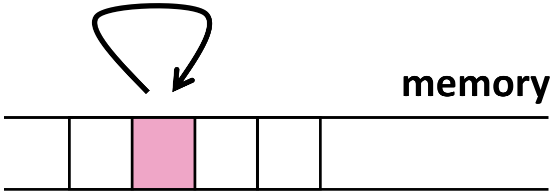
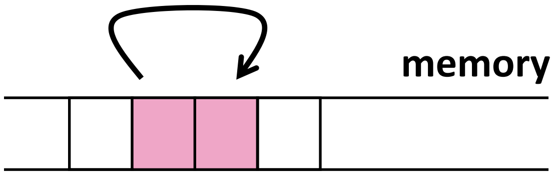
Last Time: Locality

- *Locality:* **Programs tend to use data and instructions with addresses near or equal to those they have used recently**
  History of locality

- *Temporal locality:*
  Recently referenced items are likely
  to be referenced again in the near future



- *Spatial locality:*
  Items with nearby addresses tend
  to be referenced close together in time



Last Time: Memory/Compute Bound

- **Operational intensity of a program/algorithm:**

$$I = \frac{\textit{Number of operations}}{\textit{Amount of data transferred cache} \leftrightarrow \textit{RAM}}$$

- ***"Definition:"*** **Programs with high *I* are called *compute bound*, programs with low *I* are called *memory bound***

- **Bound on operational intensity (assumes cold cache):**

$$I \leq \frac{\textit{Number of operations}}{\textit{Size of input data + size of output data}}$$

Today

- **Caches**

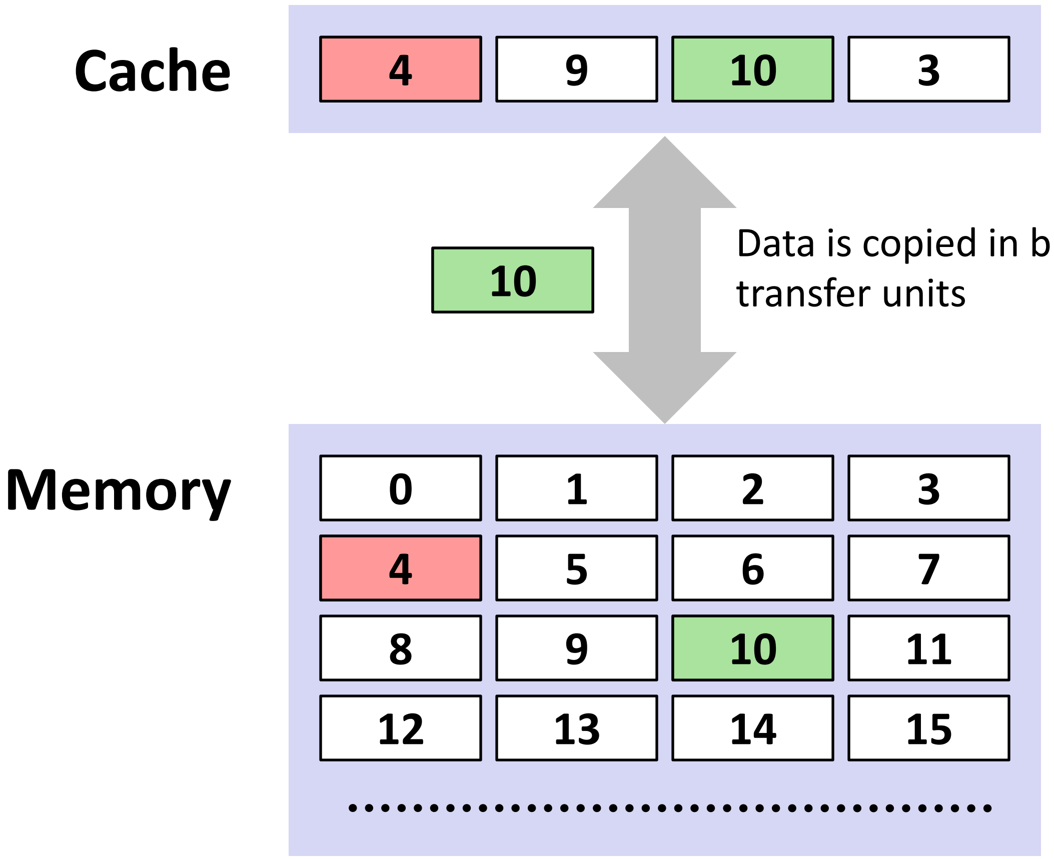
*Chapter 6 in **Computer Systems: A Programmer's Perspective**, 2nd edition, Randal E. Bryant and David R. O'Hallaron, Addison Wesley 2010*

Cache

- ***Definition:*** **Computer memory with short access time used for the storage of frequently or recently used instructions or data**

- **Naturally supports *temporal locality***
- ***Spatial locality*** **is supported by transferring data in blocks**
  - Core 2: one block = 64 B = 8 doubles

General Cache Mechanics

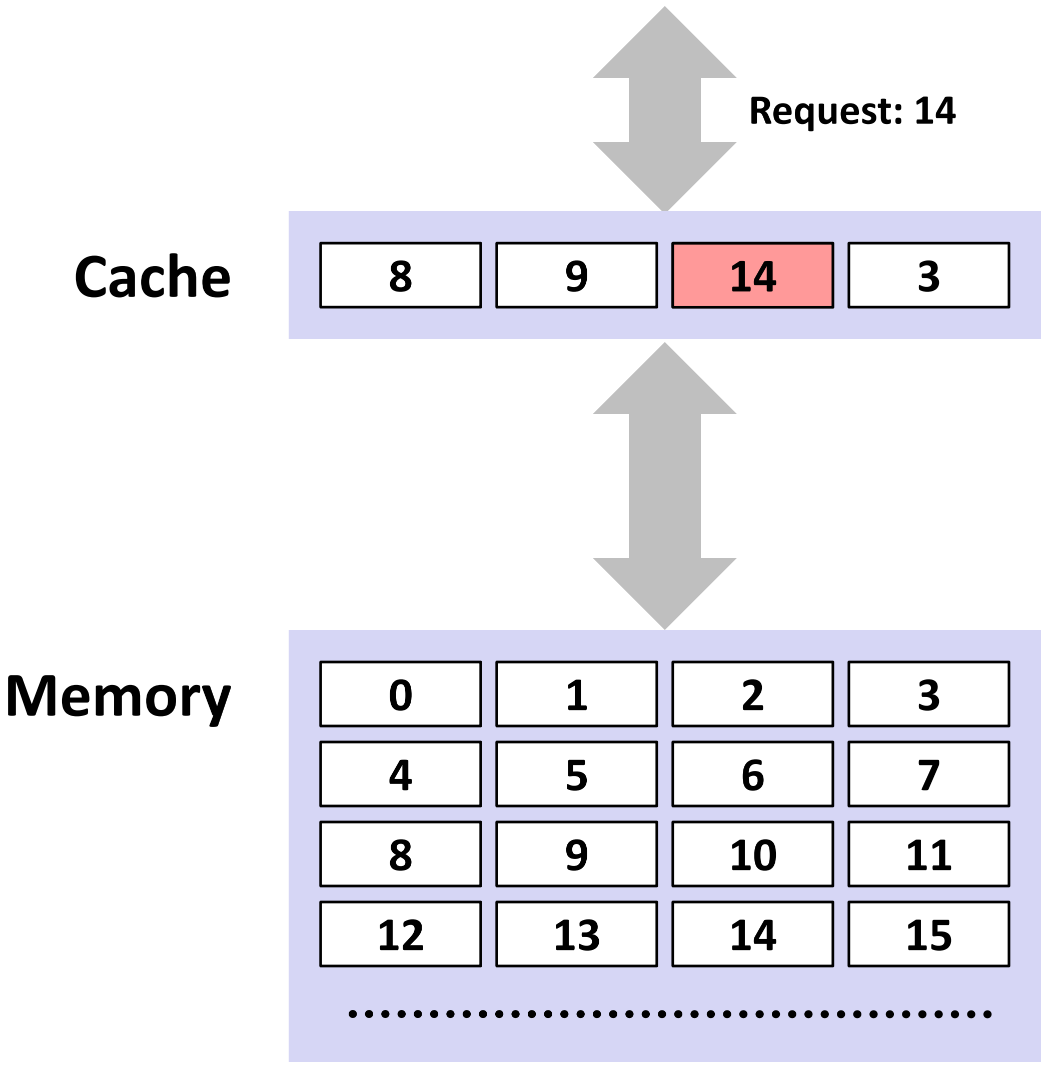


Smaller, faster, more expensive memory caches a subset of the blocks

Data is copied in block-sized transfer units

Larger, slower, cheaper memory viewed as partitioned into “blocks”

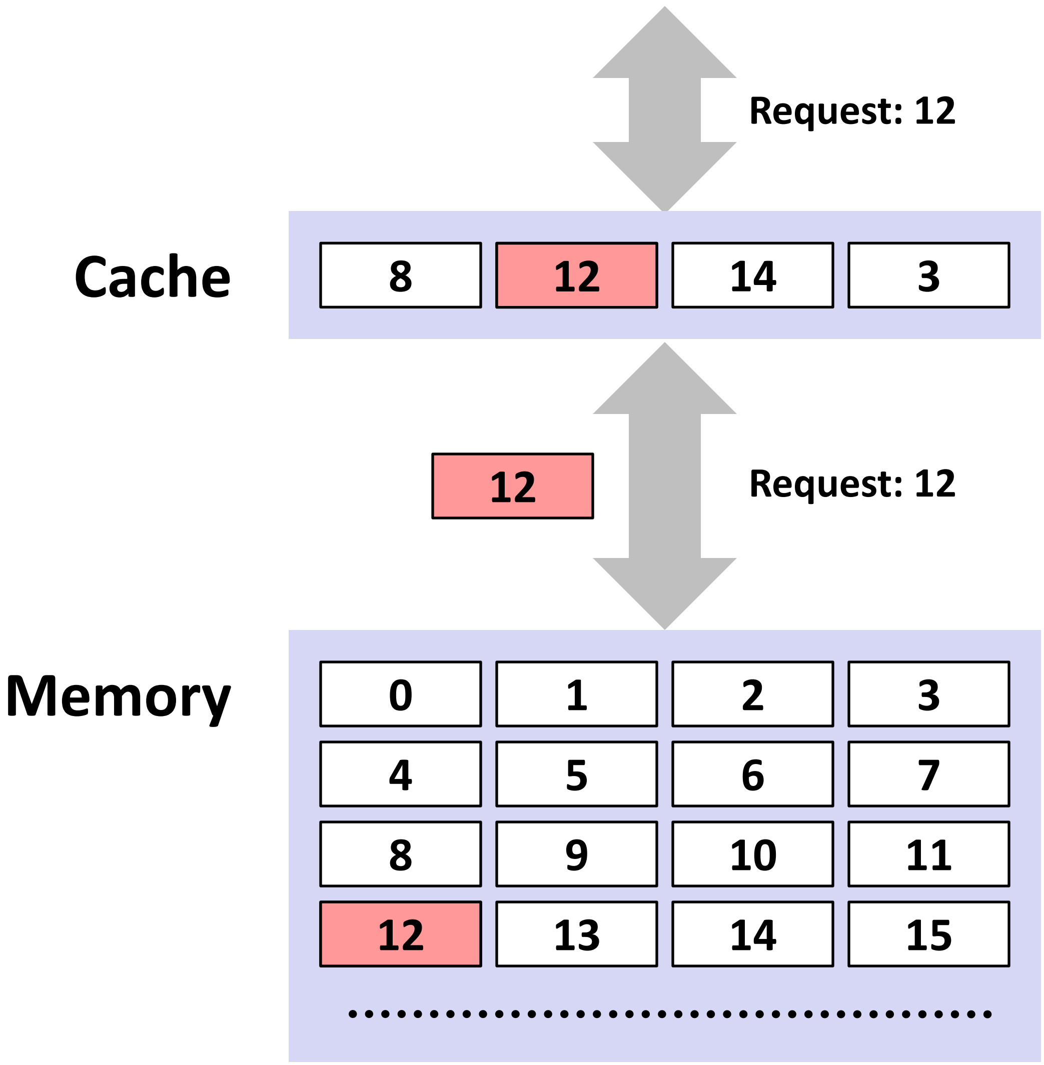
General Cache Concepts: Hit



Data in block b is needed

Block b is in cache:
Hit!

General Cache Concepts: Miss



Data in block b is needed

Block b is not in cache:
Miss!

Block b is fetched from memory

Block b is stored in cache

- *Placement policy:* determines where b goes
- *Replacement policy:* determines which block gets evicted (victim)

Types of Cache Misses (The 3 C's)

- ***Compulsory (cold)* miss**

  Occurs on first access to a block

- ***Capacity* miss**

  Occurs when working set is larger than the cache

- ***Conflict* miss**

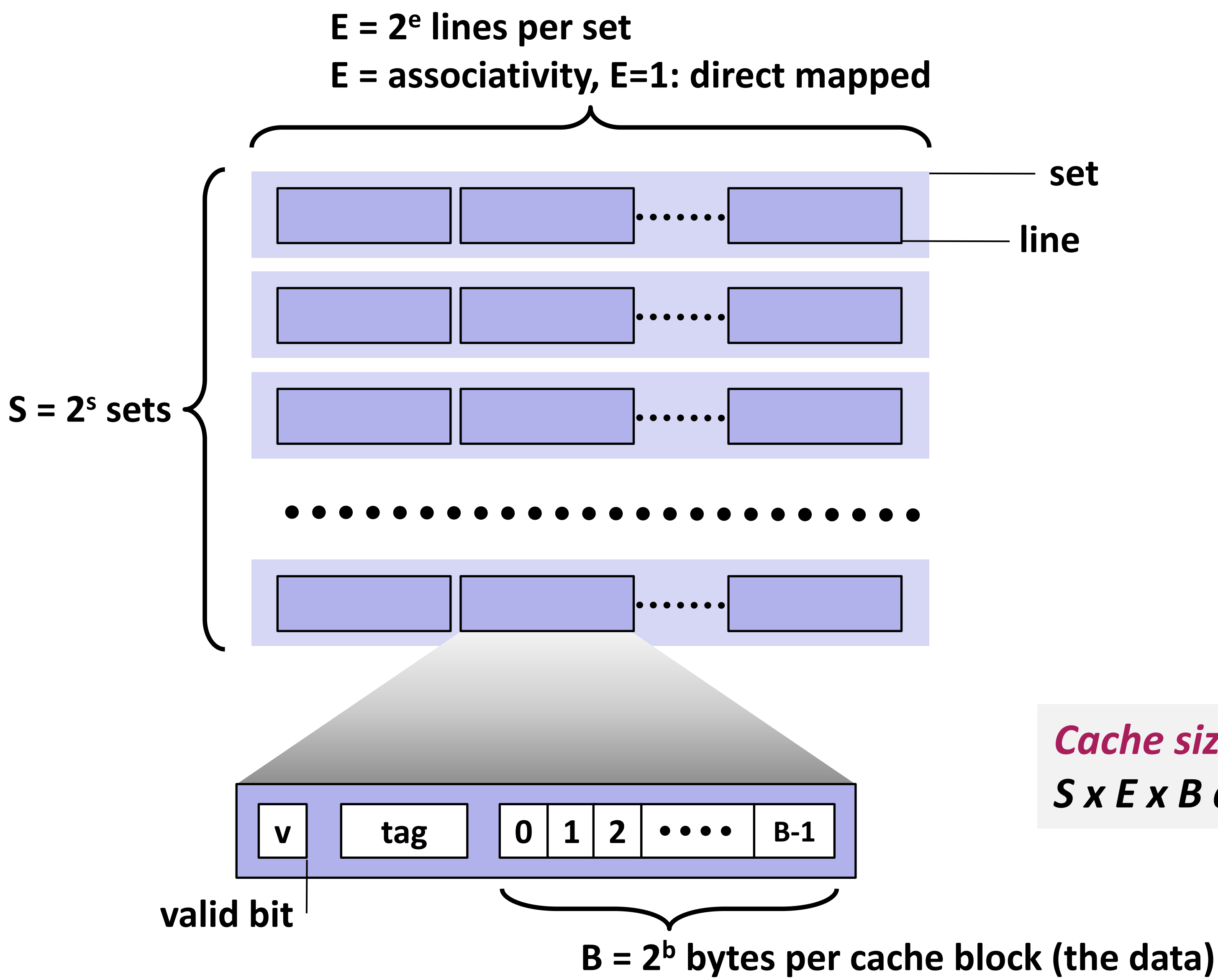
  Conflict misses occur when the cache is large enough, but multiple data objects all map to the same slot

- **Not a clean classification but still useful**

Cache Performance Metrics

- **Miss Rate**
  - Fraction of memory references not found in cache: misses / accesses
    = 1 – hit rate
- **Hit Time**
  - Time to deliver a block in the cache to the processor
  - Core 2:
    3 clock cycles for L1
    14 clock cycles for L2
- **Miss Penalty**
  - Additional time required because of a miss
  - Core 2: about 100 cycles for L2 miss

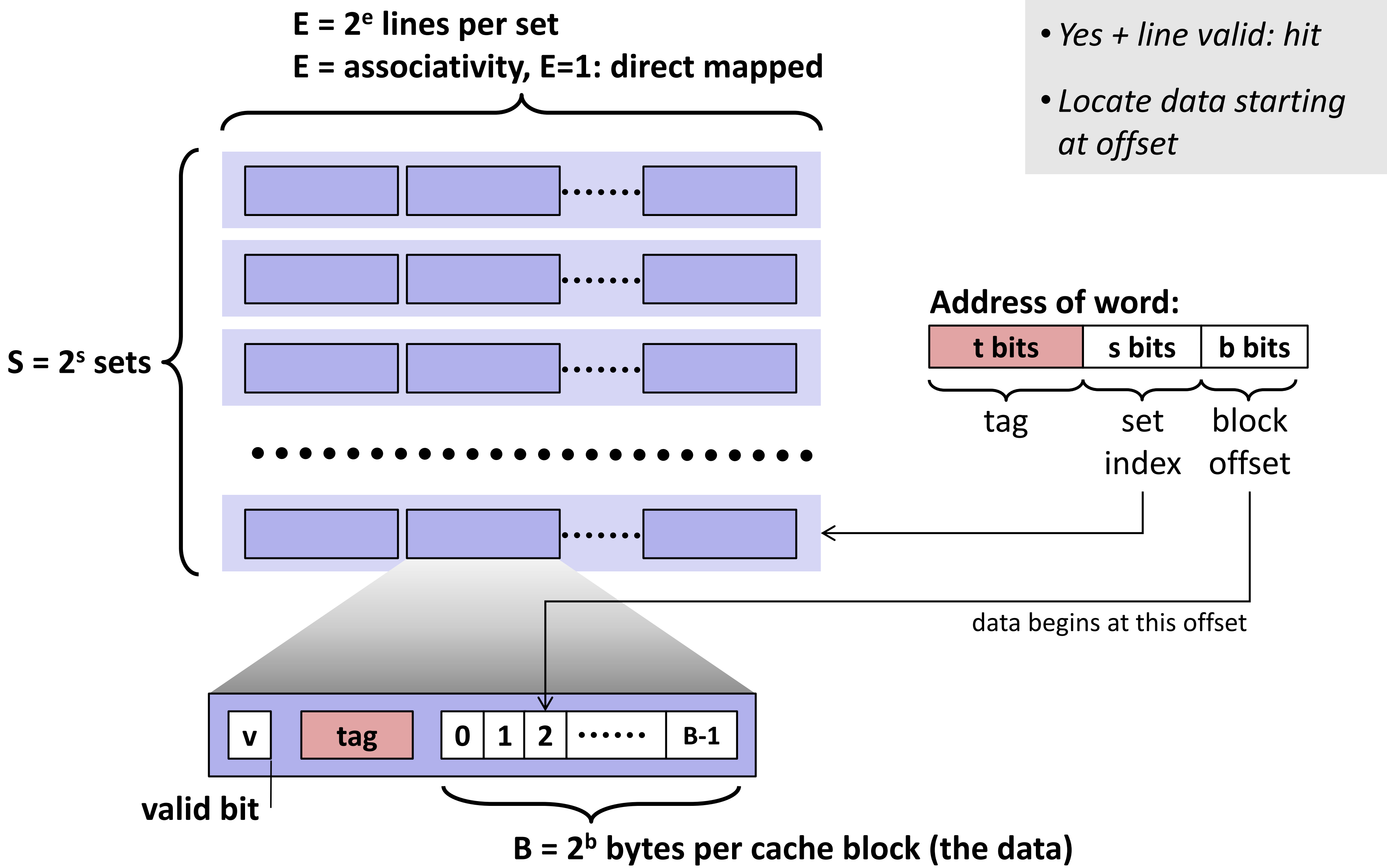
General Cache Organization (S, E, B)



Cache size:
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*



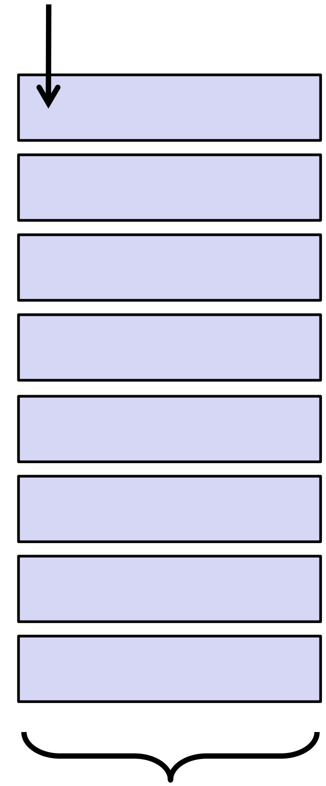
Example (S=8, E=1)

```
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;
}
```

Ignore the variables sum, i, j

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



B = 32 byte = 4 doubles

blackboard

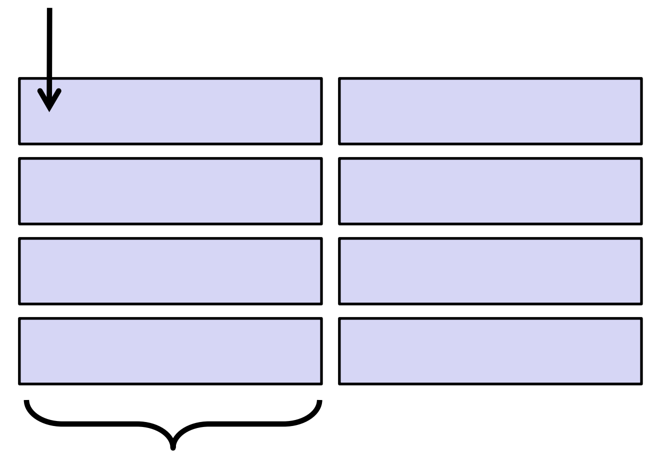
Example (S=4, E=2)

```
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;
}
```

Ignore the variables sum, i, j

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



B = 32 byte = 4 doubles

blackboard

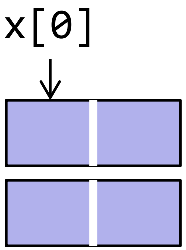
Terminology

- **Direct mapped cache:**
  - Cache with E = 1
  - Means every block from memory has a unique location in cache
- **Fully associative cache**
  - Cache with S = 1 (i.e., maximal E)
  - Means every block from memory can be mapped to any location in cache
- **LRU (least recently used) replacement**
  - when selecting which block should be replaced (happens only for E > 1), the least recently used one is chosen

What about writes?

- **What to do on a write-hit?**
  - ***Write-through:*** write immediately to memory
  - ***Write-back:*** defer write to memory until replacement of line (*needs a valid bit)*
- **What to do on a write-miss?**
  - ***Write-allocate:*** load into cache, update line in cache
  - ***No-write-allocate:*** writes immediately to memory
- **Example: (Blackboard)**
  - Example: z = x + y, x, y, z vector of length n
  - assume they fit jointly in cache
  - cold cache
- **Core 2:**
  - Write-back + Write-allocate

Small Example, Part 1



Cache:
E = 1 (direct mapped)
S = 2
B = 16 (2 doubles)

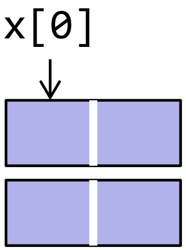
Array (accessed twice in example)
`x = x[0], …, x[7]`

```
% Matlab style code
for j = 0:1
  for i = 0:7
    access(x[i])
```

Access pattern: 0123456701234567
Hit/Miss: MHMHMHMHMHMHMHMH

Result: 8 misses, 8 hits
Spatial locality: yes
Temporal locality: no

Small Example, Part 2



Cache:
E = 1 (direct mapped)
S = 2
B = 16 (2 doubles)

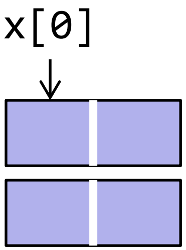
Array (accessed twice in example)
`x = x[0], …, x[7]`

```
% Matlab style code
for j = 0:1
  for i = 0:2:7
    access(x[i])
  for i = 1:2:7
    access(x[i])
```

Access pattern: 0246135702461357
Hit/Miss: MMMMMMMMMMMMMMMM

Result: 16 misses
Spatial locality: no
Temporal locality: no

Small Example, Part 3



Cache:
E = 1 (direct mapped)
S = 2
B = 16 (2 doubles)

Array (accessed twice in example)
`x = x[0], …, x[7]`

```
% Matlab style code
for j = 0:1
  for k = 0:1
    for i = 0:3
      access(x[i+4j])
```

Access pattern: 0123012345674567
Hit/Miss: MHMHHHHHMHMHHHHH

Result: 4 misses, 8 hits (is optimal, why?)
Spatial locality: yes
Temporal locality: yes

Locality Optimization: Blocking

- **Example: MMM (blackboard)**

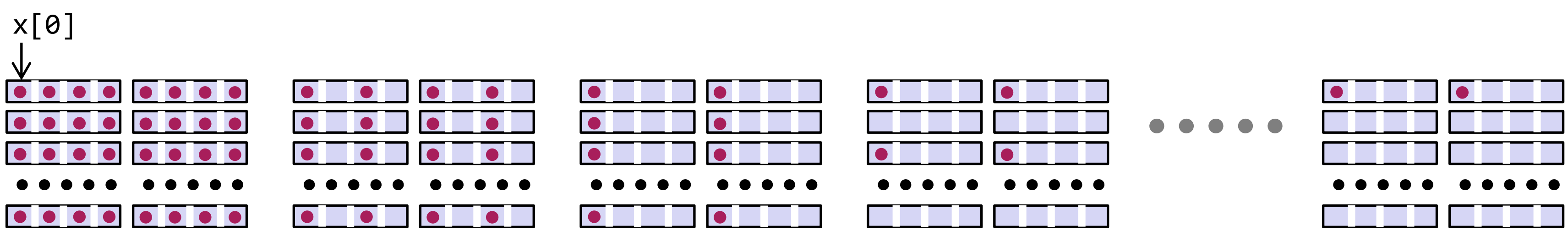
The Killer: Two-Power Strided Access

blackboard

```
% Matlab style code
% x = x[0], …, x[n-1], n >> cache size
% t = 1,2,4,8,… a 2-power
for i = 0:(n/t)
  access(x[t*i])
```

Cache: E = 2, B = 4 doubles

t = 1: **t = 2:** **t = 4:** **t = 8:** **t ≥ 4S:**



t = 1: Spatial locality
Full cache used

t = 2: Some spatial locality
1/2 cache used

t = 4: No spatial locality
1/4 cache used

t = 8: No spatial locality
1/8 cache used

t ≥ 4S: No spatial locality
1/(4S) of cache used

The Killer: Where Does It Occur?

- **Accessing two-power size 2D arrays (e.g., images) columnwise**
  - 2d Transforms
  - Stencil computations
  - Correlations
- **Various transform algorithms**
  - Fast Fourier transform
  - Wavelet transforms
  - Filter banks