Cache Memories

李舒辰

2019 年 10 月 31 日

Outline

1. Locality
2. Memory Hierarchies
3. Cache Memories
4. The Impact of Caches on Program Performance

Locality

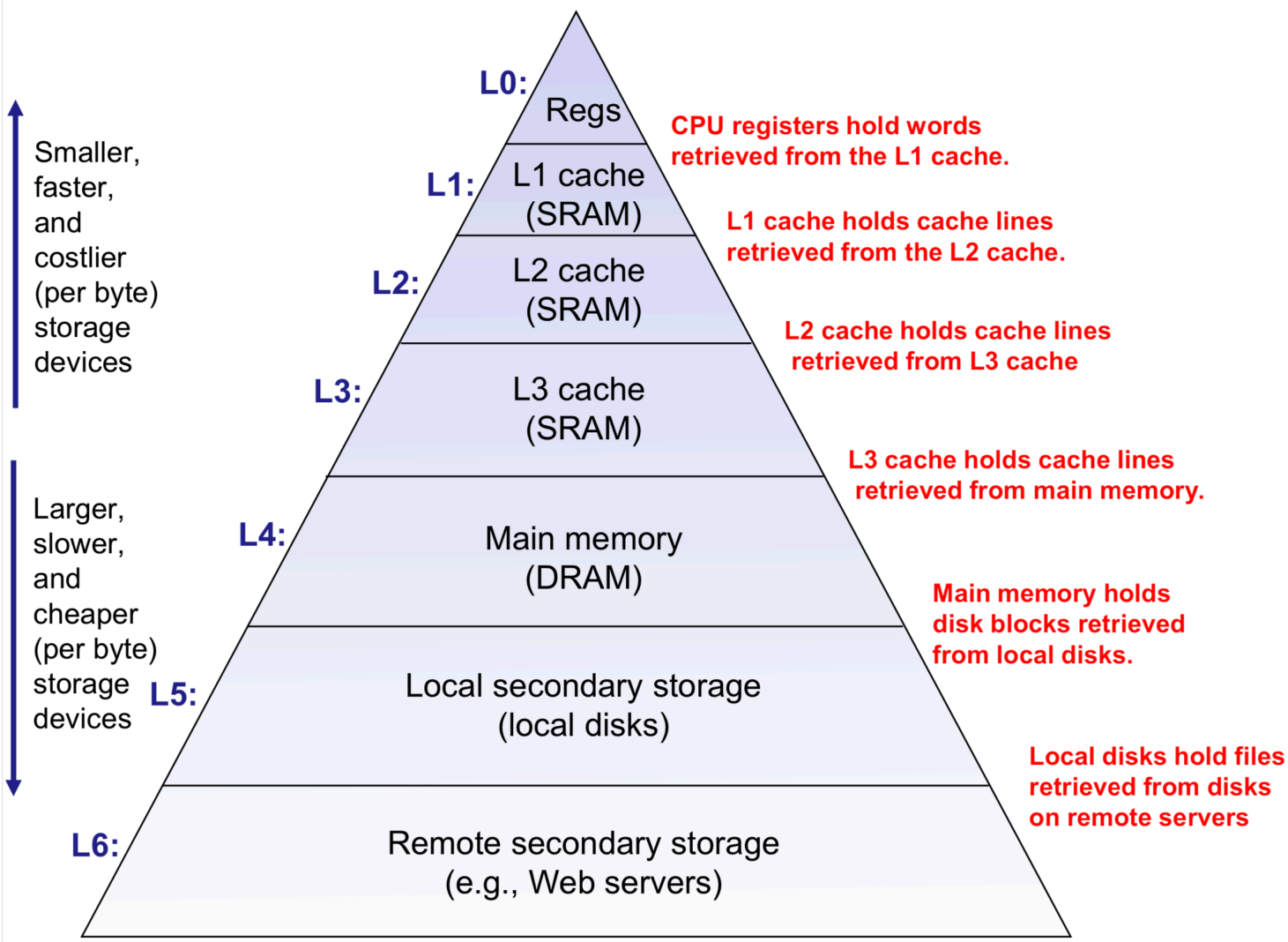
Locality

Programs tend to use data and instructions with addresses near or equal to those they have used recently.

- Temporal
- Spatial
- Program data
- Instructions

Memory Hierarchies

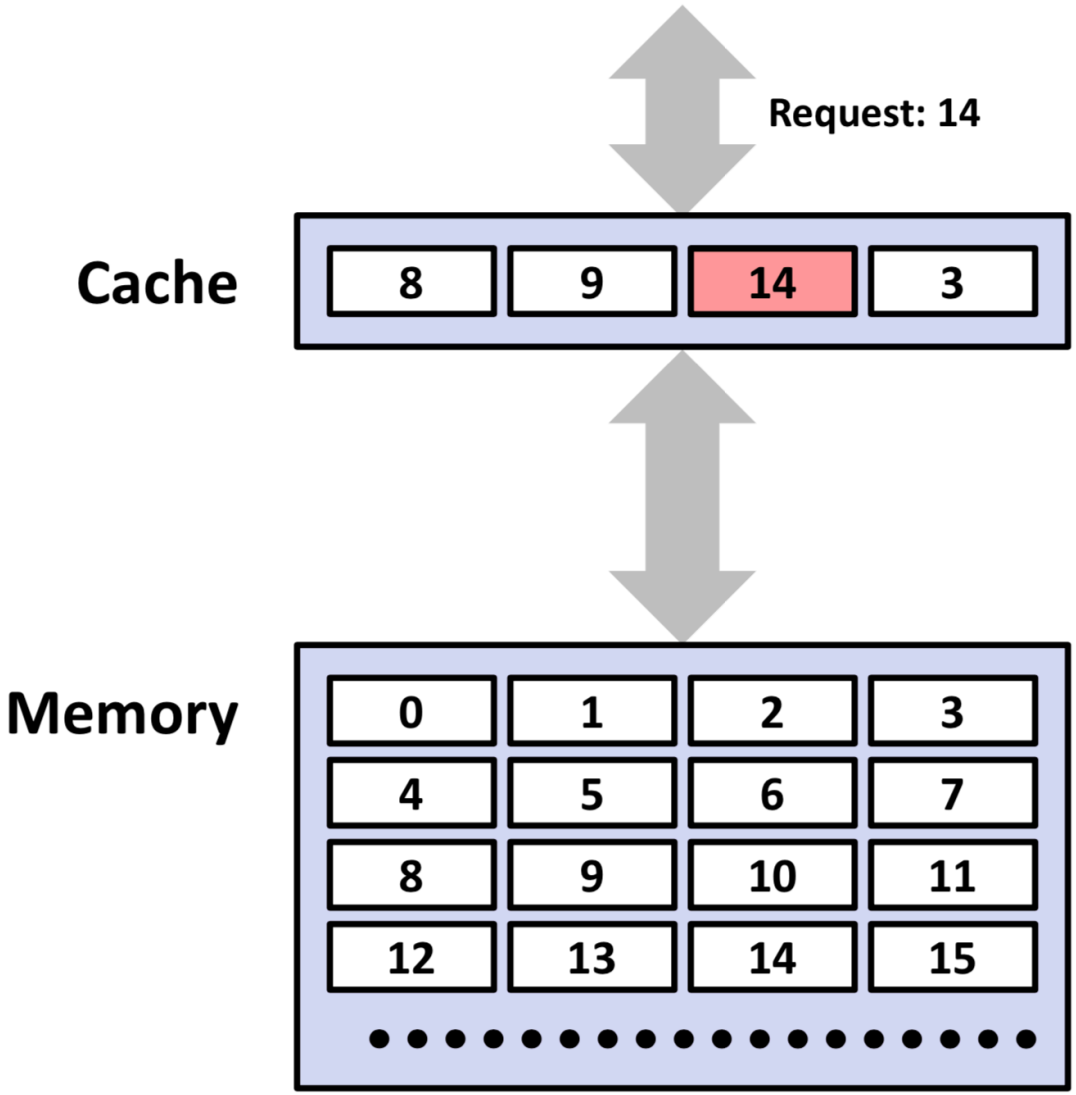
Memory Hierarchies



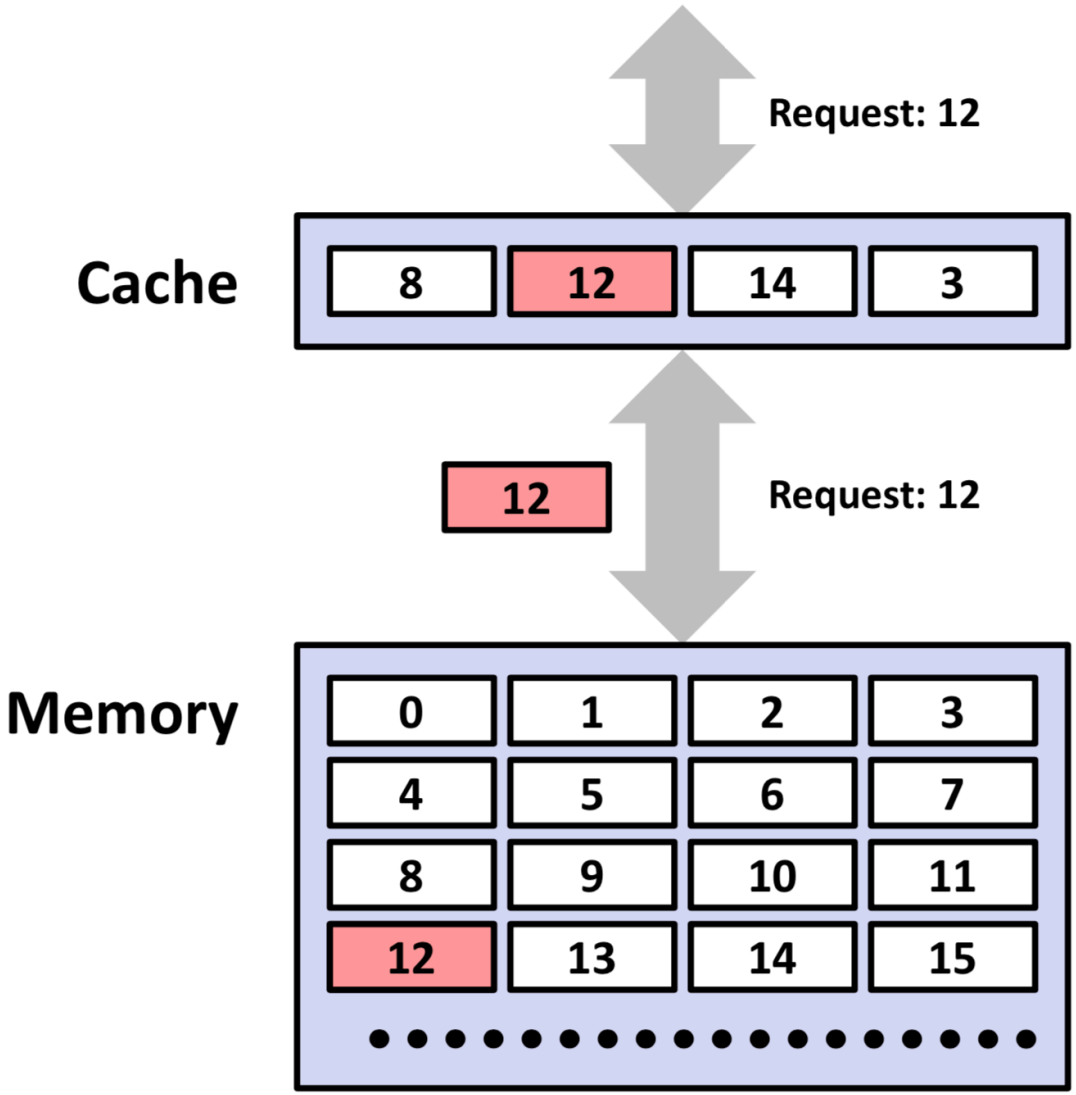
Cache

- Cache: small, fast, a staging area for the data objects stored in a larger, slower device.
- The faster, smaller storage device at level k serves as a cache for the slower, larger device in level $k+1$.
- Be partitioned into a smaller set of blocks (the same size as that at next level).
  - Contains copies of a subset of the blocks from level $k+1$.
  - Be copied in block-size transfer units.

Cache: Hit



Cache: Miss


Request: 12
Cache
8
12
14
3
12
Request: 12
Memory
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Cache: Miss

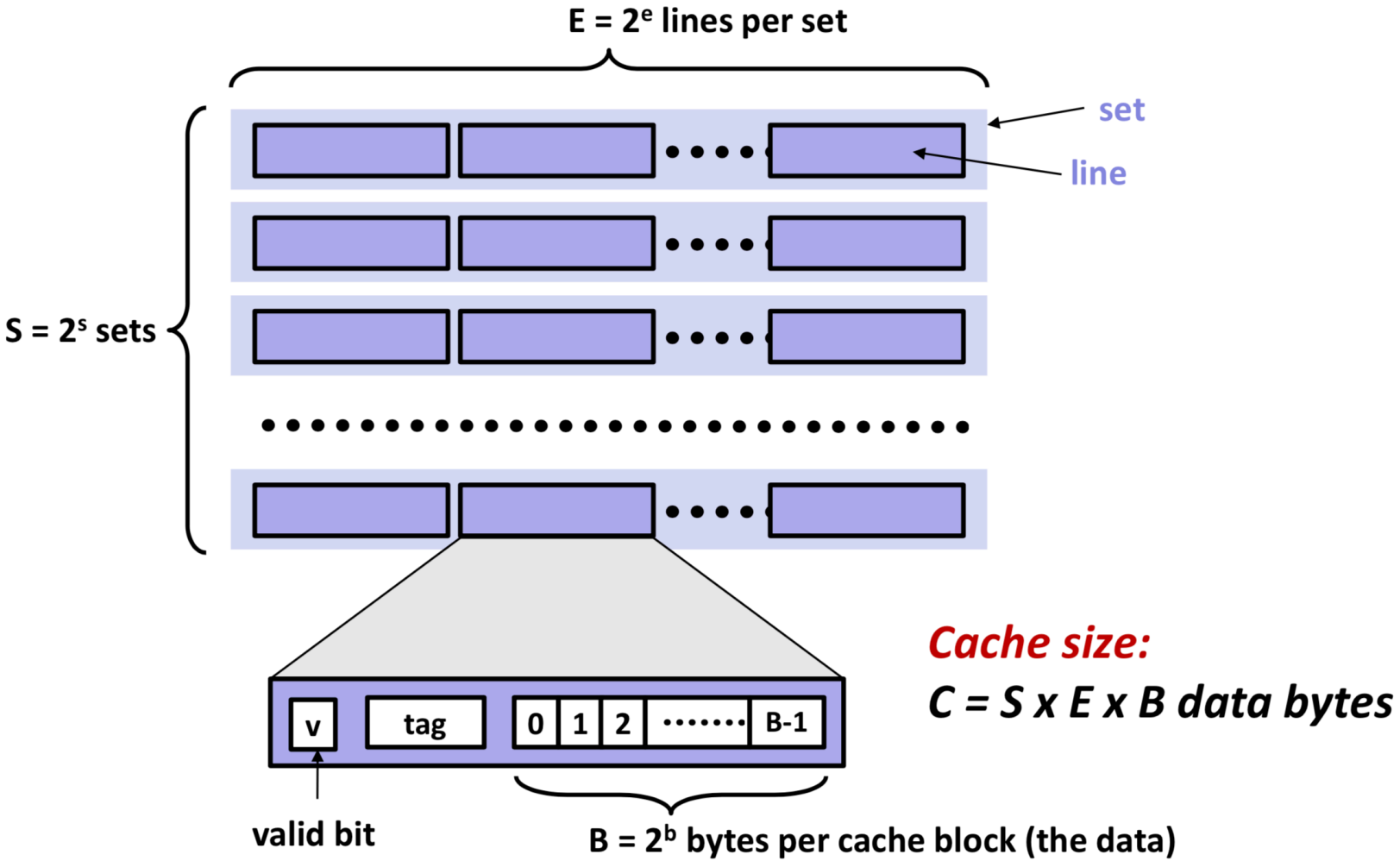
- Placement policy
  - Random placement policy: expensive to locate blocks
  - Restrict a particular block at level $k+1$ to a small subset (singleton) of the blocks at level k : e.g. $i \mapsto i \bmod 4$
- Replacement policy
  - Random replacement policy
  - Least Recently Used (LRU)

Types of cache misses

- Cold (compulsory) miss
- Conflict miss: occurs when the cache is large enough, but multiple objects map to the same block.
- Capacity miss: occurs when the size of working set exceeds the size of the cache.

Cache Memories

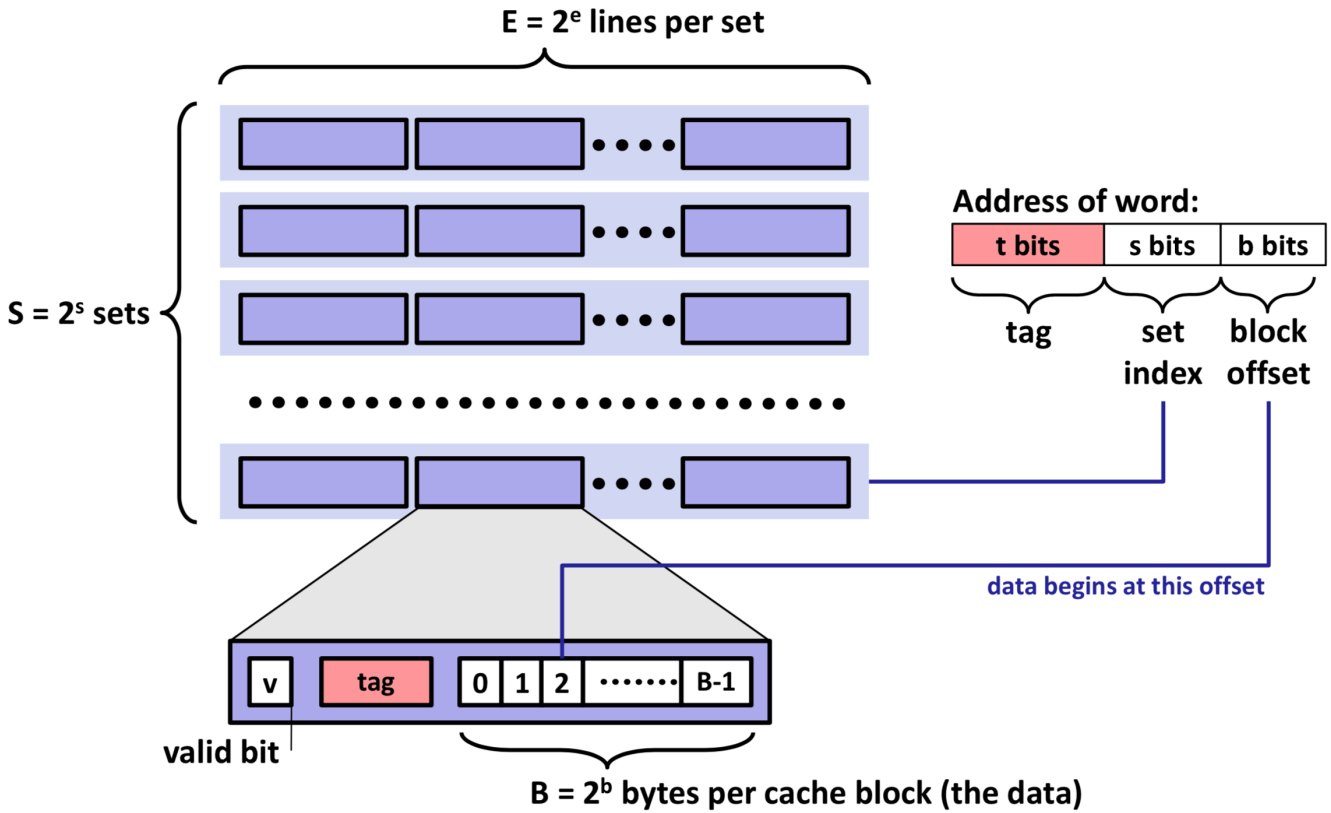
Cache Organization



Cache size:

C = S x E x B data bytes

Cache Read



1. Locate set
2. Matching tag + line valid: Hit
3. locate data starting at offset

Direct Mapped Cache ($E = 1$)

1. Set selection
2. Line matching
3. Word selection
4. Line replacement

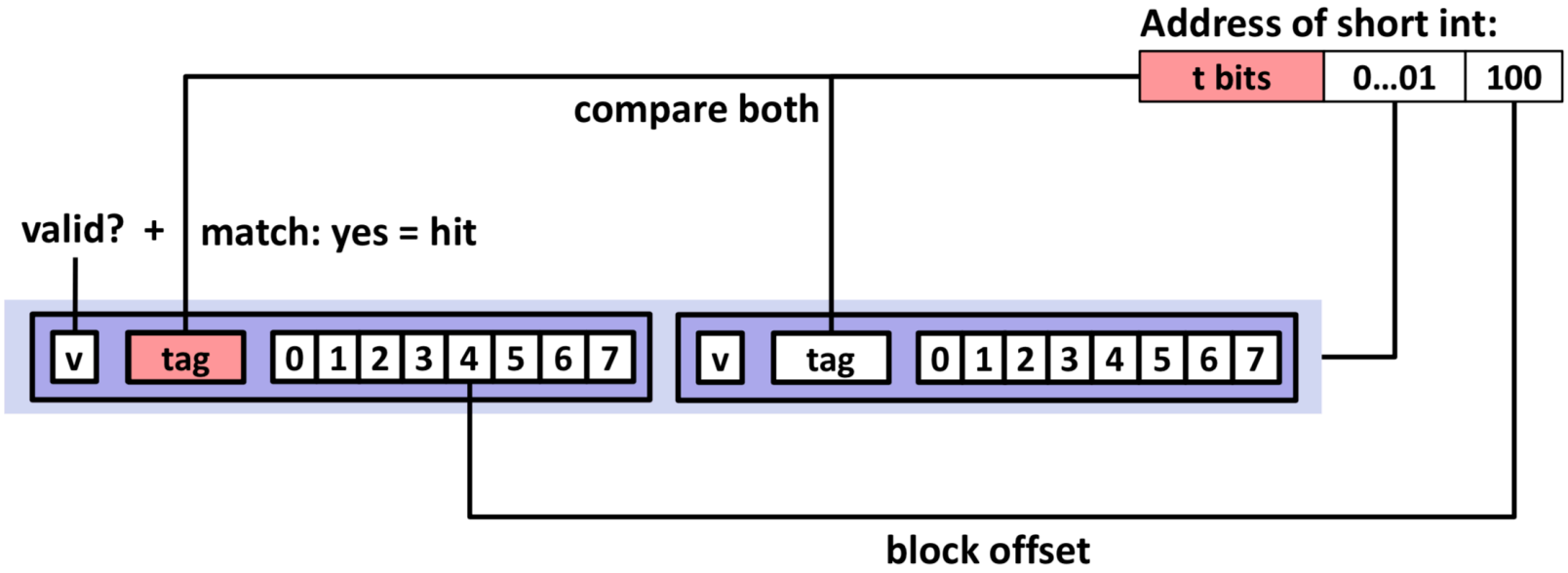
Direct Mapped Cache: Conflict Miss

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

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

元素	地址	组索引	元素	地址	组索引
x[0]	0	0	y[0]	32	0
x[1]	4	0	y[1]	36	0
x[2]	8	0	y[2]	40	0
x[3]	12	0	y[3]	44	0
x[4]	16	1	y[4]	48	1
x[5]	20	1	y[5]	52	1
x[6]	24	1	y[6]	56	1
x[7]	28	1	y[7]	60	1

E-way Set Associative Cache



- Line matching: associative storage, key = tag + valid bits, value = block contents
- Line replacement:
  - Random
  - Least Frequently Used (LFU)
  - LRU

Issues with Writes

- Write-hit
  - Write-through: write immediately to the next level
  - Write-back: write to memory until replacement
    - dirty bit
- Write-miss
  - Write-allocate: load into cache, update the block
  - No-write-allocate: write straight to the next level

Cache Performance Metrics

- Miss rate: #misses / #references = 1 – hit rate
- Hit time: the time to deliver a word in the cache to CPU
  - On the order of several clock cycles for L1 caches
- Miss penalty: additional time required because a miss
  - The penalty for L1 misses served from L2 is on the order of 10 cycles; from L3, 50 cycles; from main memory, 200 cycles

Writing Cache Friendly Code

Minimize the misses:

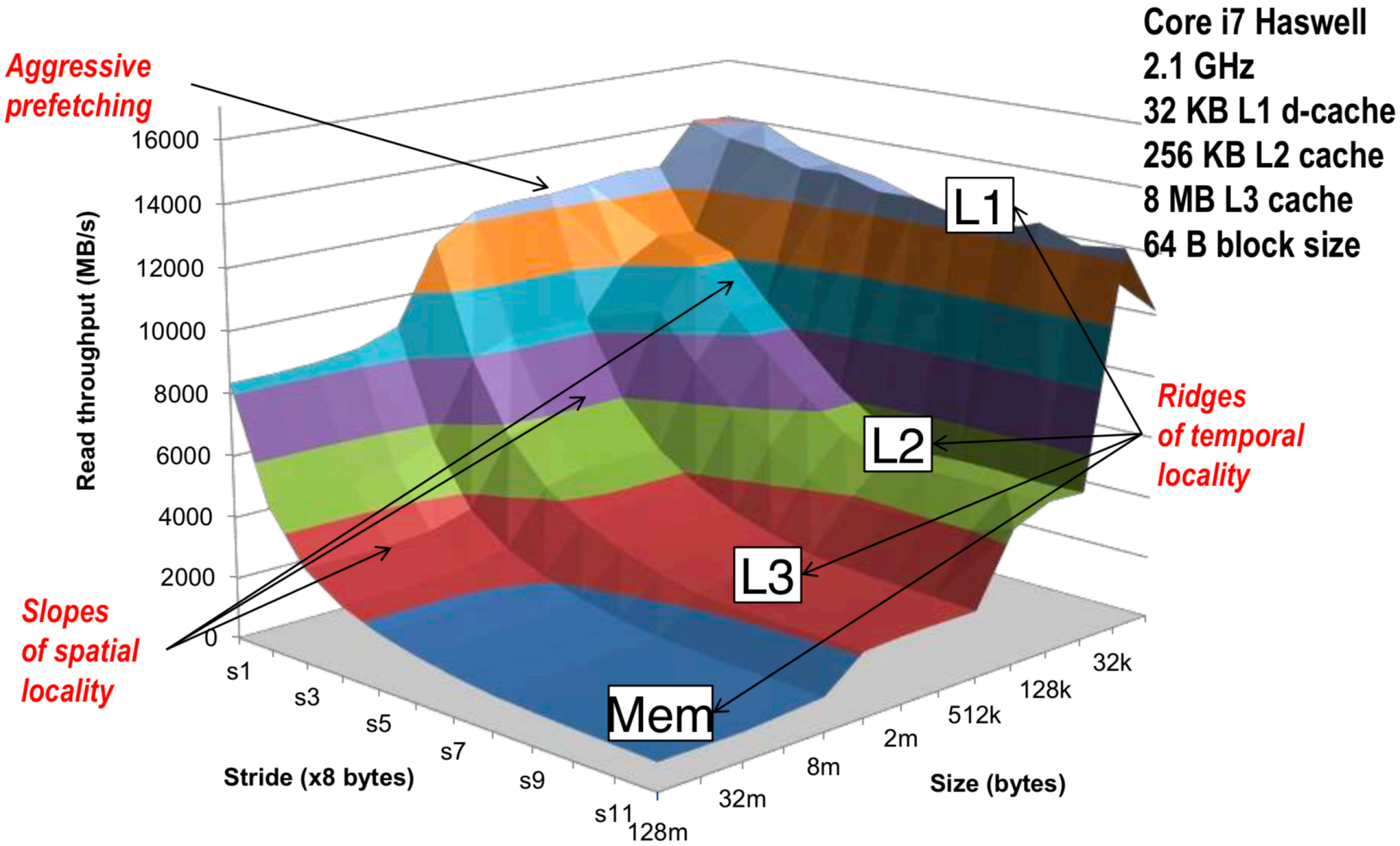
- Repeated references to variables (temporal locality)
- Stride-1 reference patterns (spatial locality)

The Impact of Caches on Program Performance

The Memory Mountain

- Read throughput (read bandwidth): number of bytes read from memory per second. (MB/s)
- Measure read throughput: memory mountain
  - a function of spatial and temporal locality

The Memory Mountain



Rearranging Loops to Improve Spatial Locality

Matrix multiplication: $\mathbf{C}_{n\times n} = \mathbf{A}_{n\times n}\mathbf{B}_{n\times n}$

- Elements are doubles (8 bytes)
- $\Theta(n^3)$
- n is so large that a single row does not fit in the L1 cache
- Block size $B = 32$ byte

Analysis method: look at access pattern of inner loop.

Inner Loop

- Stepping through columns in one row
  - `for (i = 0; i < n; i++) sum += a[0][i];`
  - exploit spatial locality (assuming element size $<$ block size)
  - miss rate $=$ element size / block size $= 8/32 = 0.25$
- Stepping through rows in one column
  - `for (i = 0; i < n; i++) sum += a[i][0];`
  - no spatial locality (assuming n is very large)
  - miss rate $= 1$

Summary of Matrix Multiplication

```
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;
 }
}
```

ijk (& jik):

- 2 loads, 0 stores
- misses/iter = **1.25**

```
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];
 }
}
```

kij (& ikj):

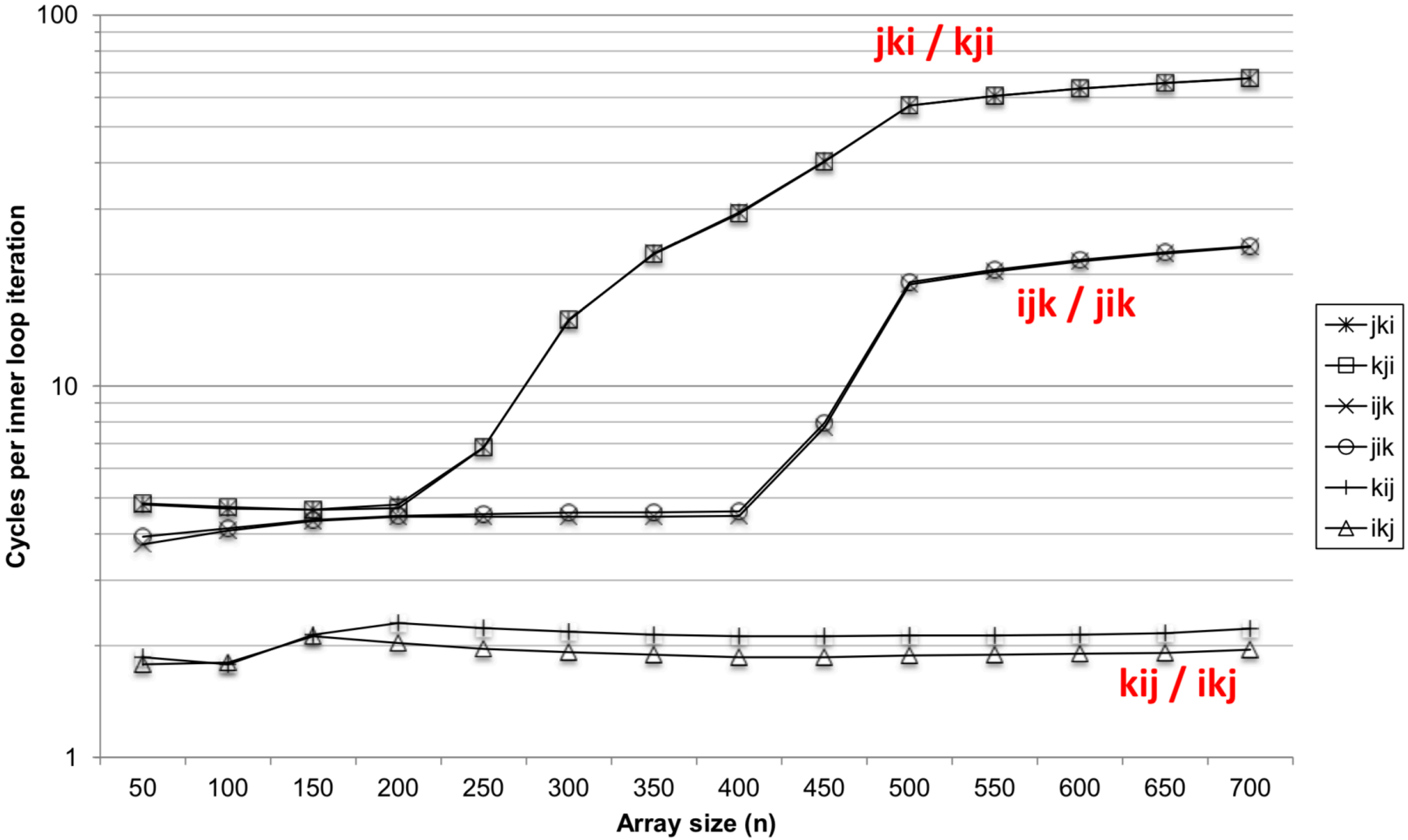
- 2 loads, 1 store
- misses/iter = **0.5**

```
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;
 }
}
```

jki (& kji):

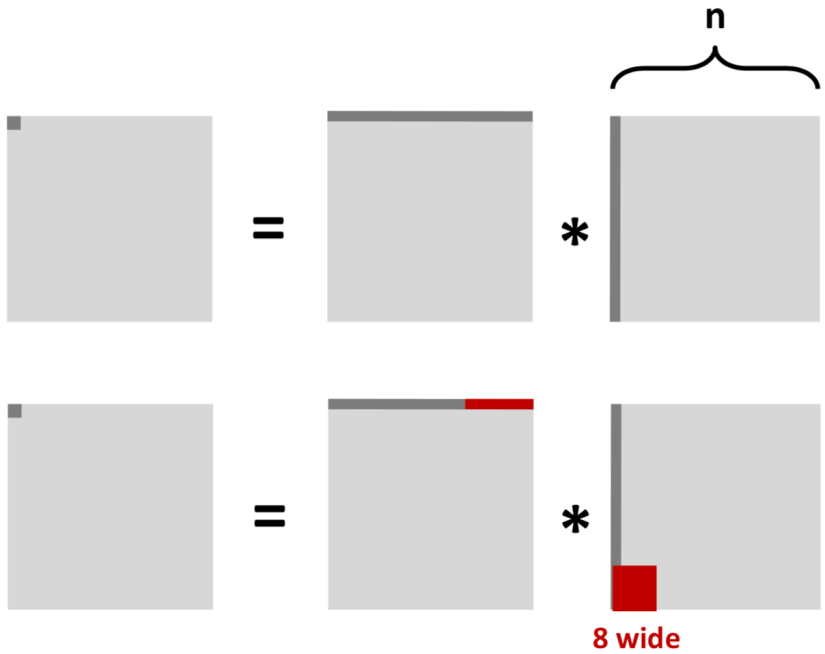
- 2 loads, 1 store
- misses/iter = **2.0**

Core i7 Matrix Multiplication Performance



Using Blocking to Improve Temporal Locality

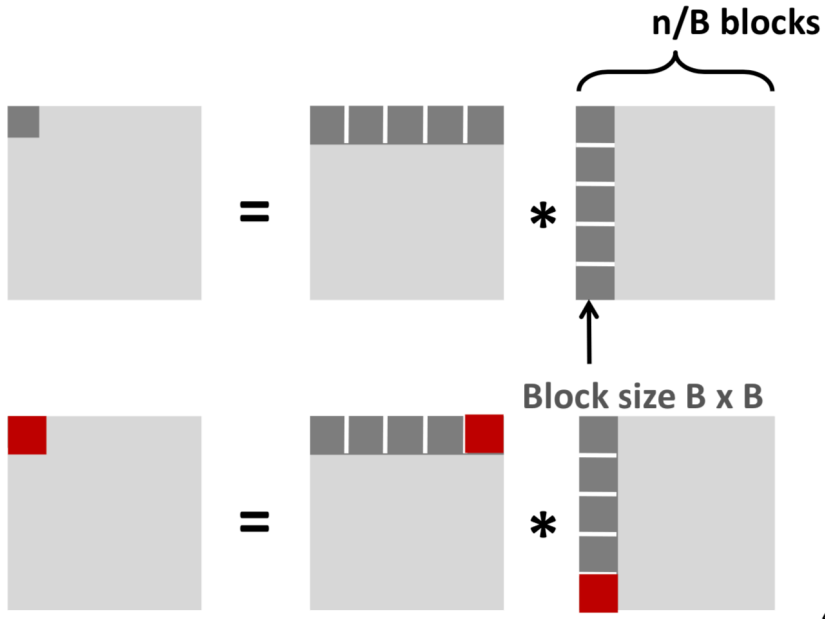
- Block size = 8 doubles



- First iteration: $n/8 + n = 9n/8$ misses. (The rest iterations are the same)
- Total misses: $9n/8 \cdot n^2 = 9/8 \cdot n^3$

Using Blocking to Improve Temporal Locality

- Three blocks fit into cache



- First iteration: $B^2/8 \cdot 2n/B = nB/4$ misses. (The rest iterations are the same)
- Total misses: $nB/4 \cdot (n/B)^2 = 1/4B \cdot n^3$