[CSED211] Introduction to Computer Software Systems

Lecture 11: Cache Memory

Prof. Jisung Park

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

CAOS
COMPUTER ARCHITECTURE & OPERATING SYSTEMS LABORATORY

2023.11.08

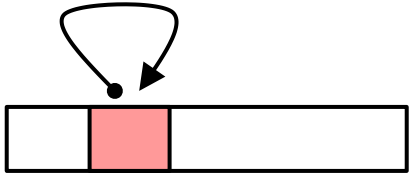
Lecture Agenda

- Cache Memory Organization and Operation
- Performance Impact of Caches
  - The Memory Mountain
  - Rearranging Loops to Improve Spatial Locality
  - Using Blocking to Improve Temporal Locality

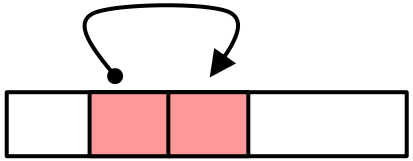
Recall: Locality

- Principle of Locality: programs tend to use data and instructions with addresses near or equal to those they have used recently

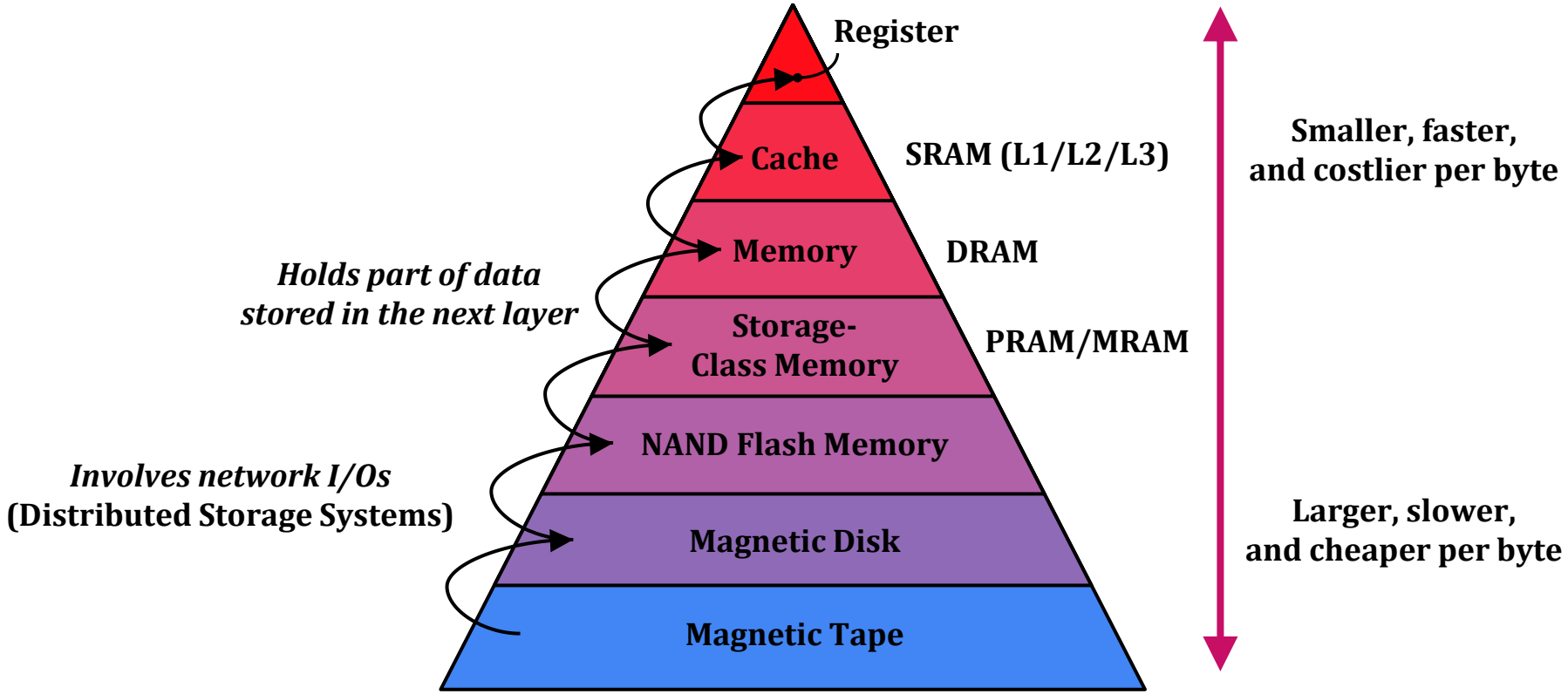
- Temporal locality:
  - Recently referenced items are likely to be referenced again in the near future



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

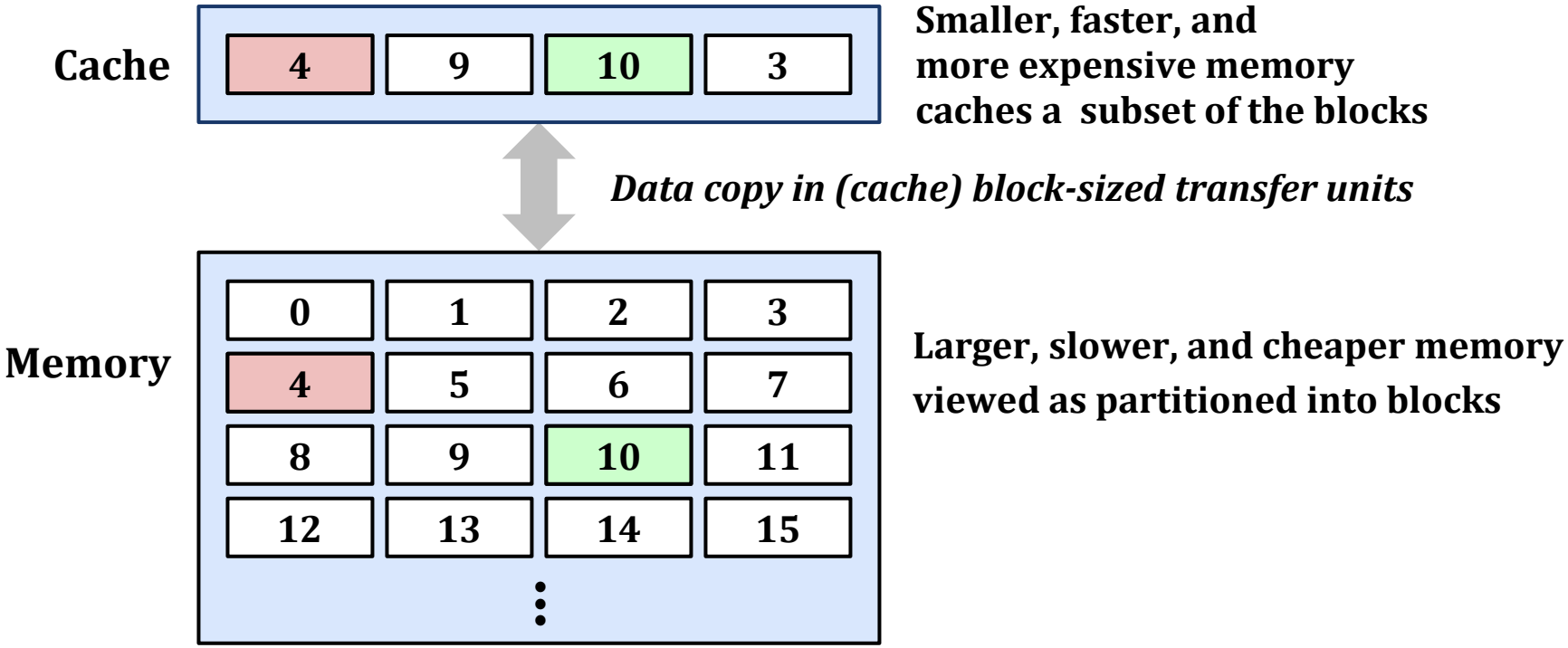


Example Memory Hierarchy

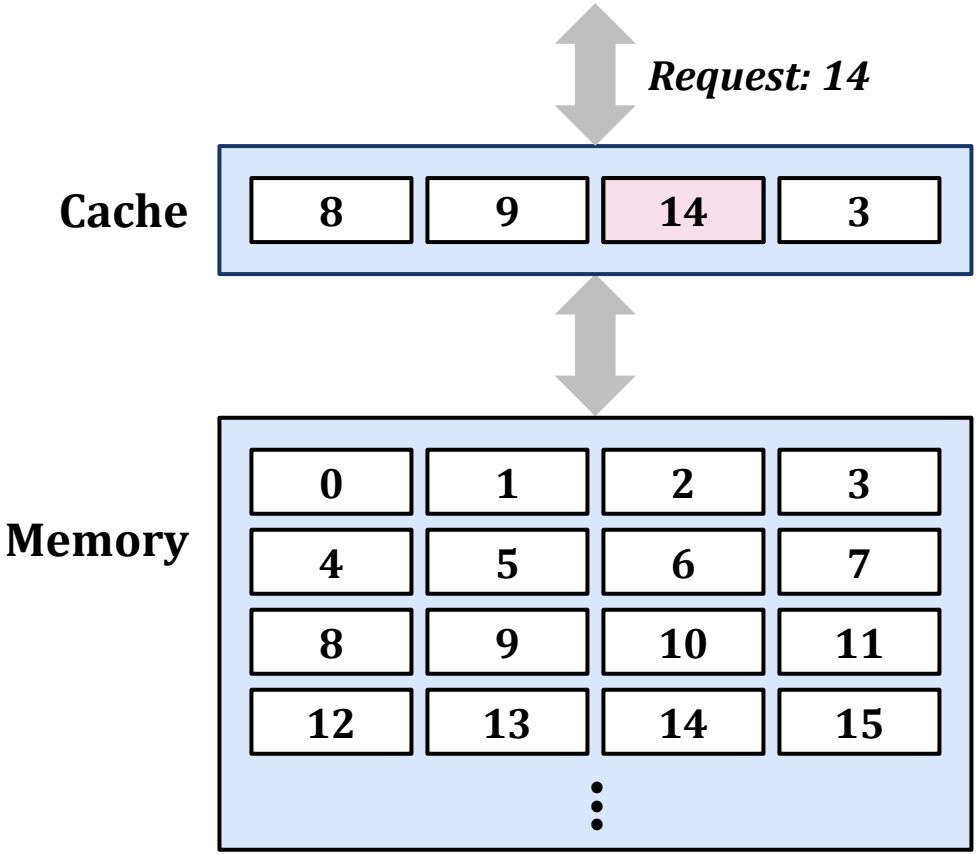


Memory Hierarchy

General Cache Concepts



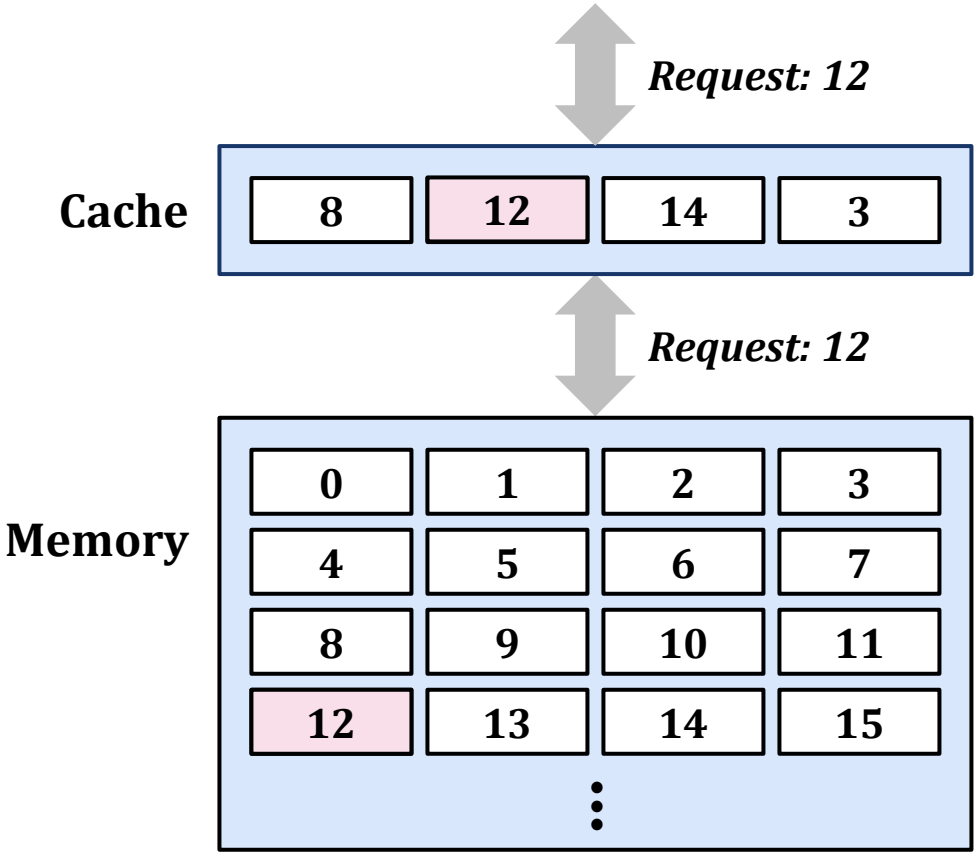
General Cache Concepts: Hit



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

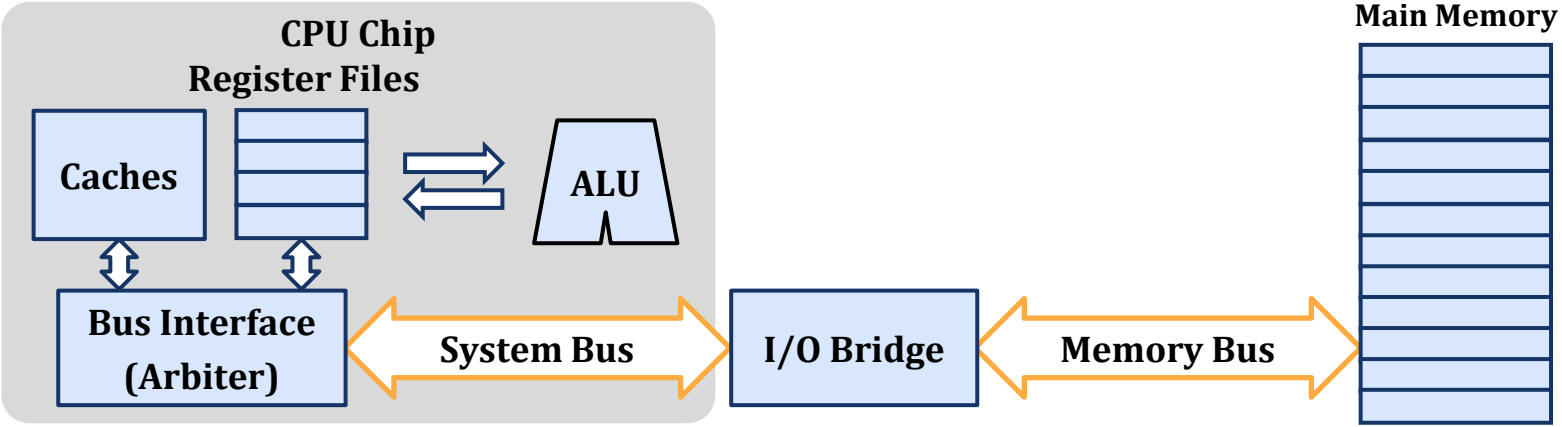
Replacement policy:
determines which block gets evicted (victim)

General Caching Concepts: 3 Types of Cache Misses

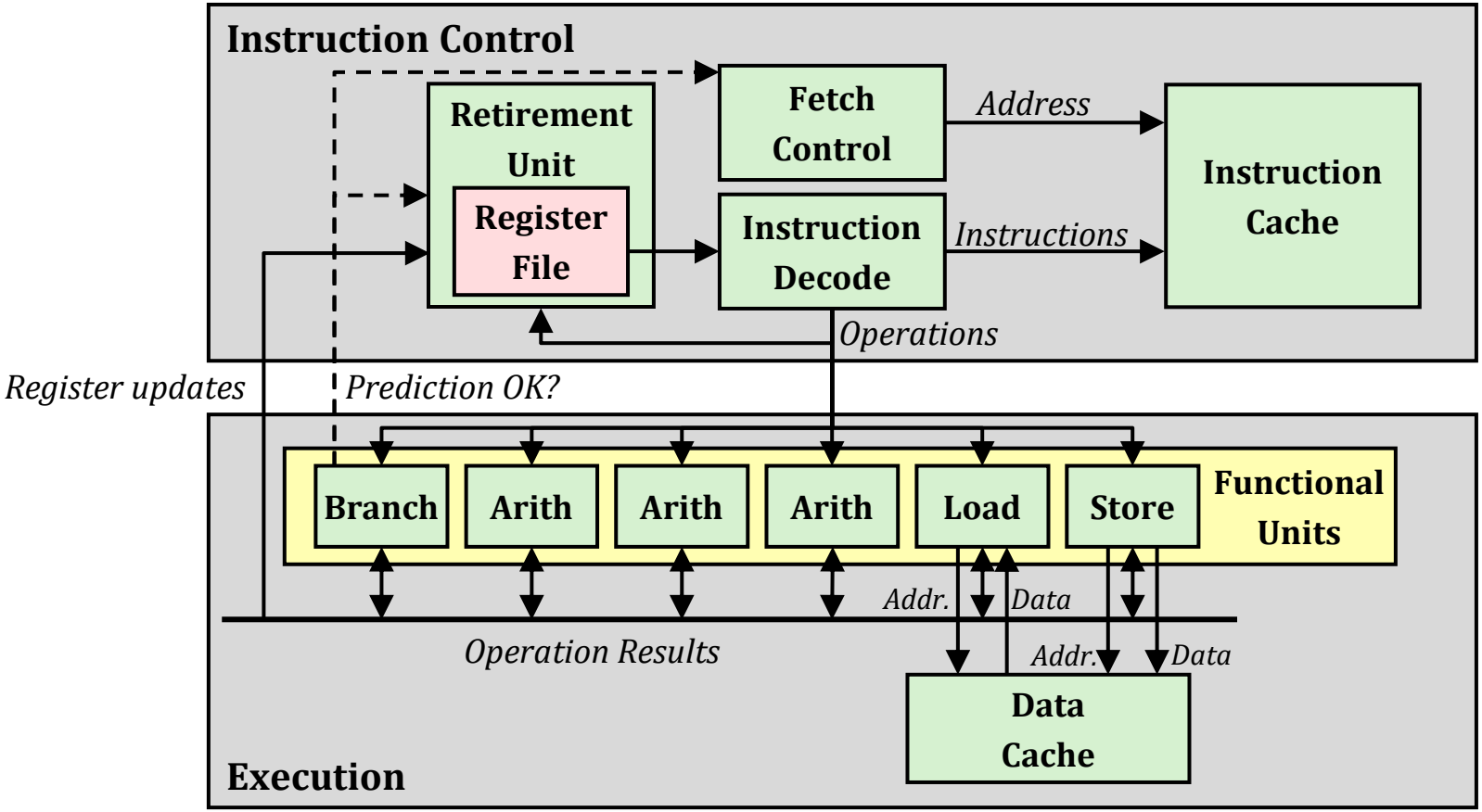
- Compulsory miss (or cold miss)
  - Occurs because any cache starts empty: the first reference to the block
- Capcity miss
  - Occurs when the set of active cache blocks (working set) is larger than the cache
- Conflict miss
  - Occurs when the level-k cache is large enough, but multiple data objects all map to the same level-k block
    - Most caches limit blocks at level (k+1) to a small subset (sometimes a singleton) of the block positions at level k – set associative cache
  - e.g., Referencing blocks 0 → 8 → 0 → 8 → 0 → 8 → … would miss every time if block i at level (k+1) must be placed in block (i mod 4) at level (k+1)

Cache Memory

- Cache memory is small, fast SRAM-based memory
  - Which is automatically managed in hardware
  - Holds frequently accessed blocks of main memory
- CPU first looks for data in caches (e.g., L1, L2, and L3), then in main memory



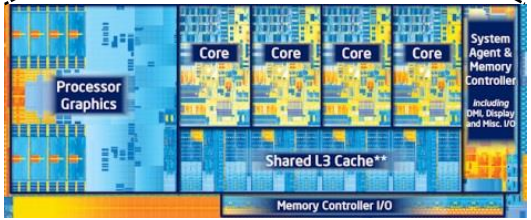
Recall: Modern CPU Design

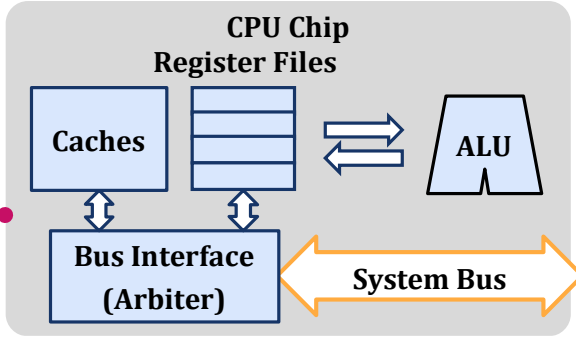


What It Really Looks Like

Desktop PC

CPU



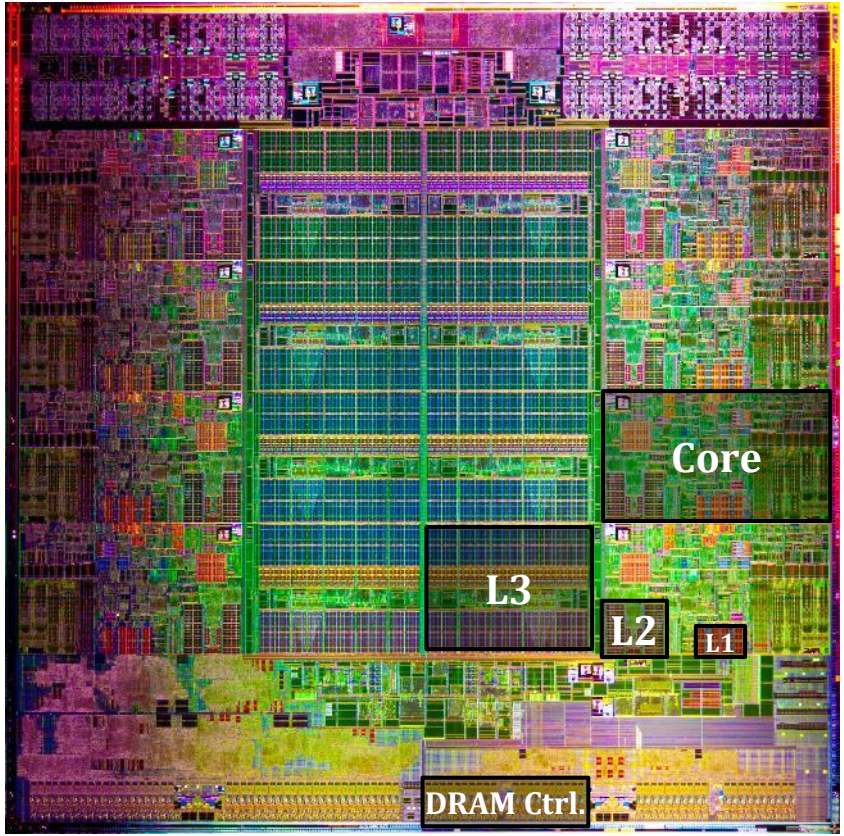


Motherboard

DRAM

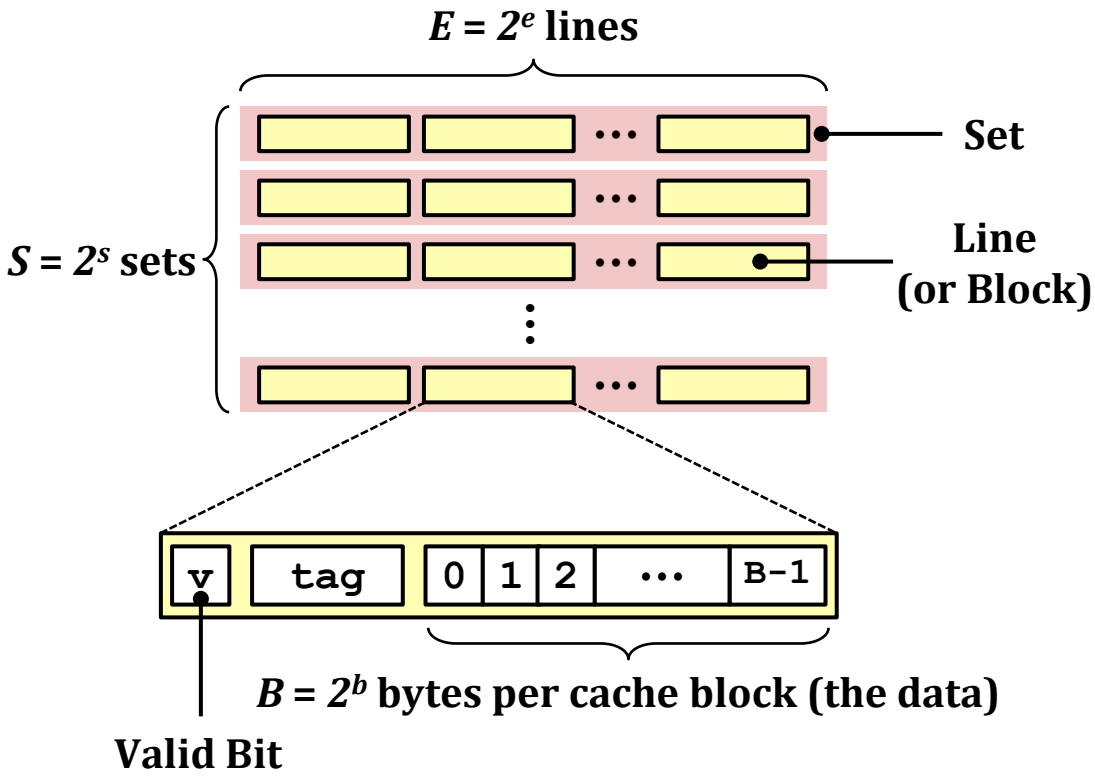
What It Really Looks Like (Cont.)

- Intel Sandy Bridge processor die
  - L1: 32-KB instruction + 32-KB data
  - L2: 256 KB
  - L3: 3~20MB



General Cache Organization

- Cache size C = S × E × B
  - S: # of sets in cache
  - E: # of lines per set
  - B: cache line (or block) size

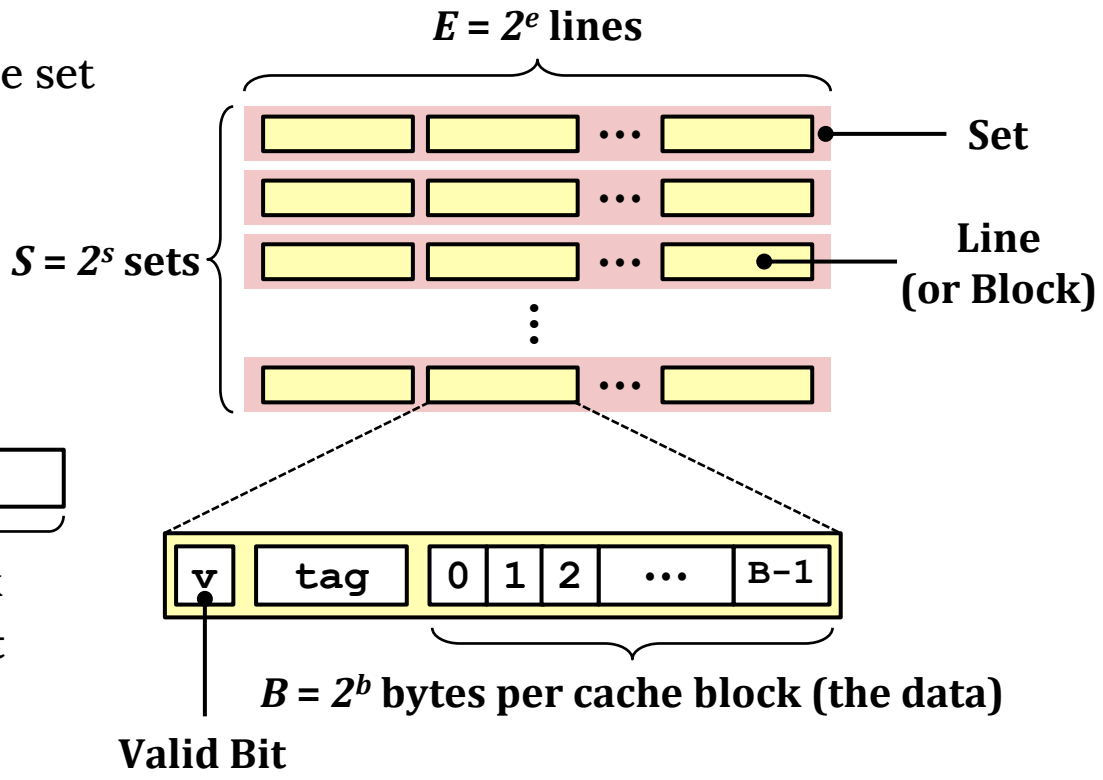


Cache Read

- Five steps of cache-line read
  - Locate the set
  - Check if any line exists in the set
  - Check if the tag matches
  - Check the valid bit
  - Locate data starting at the offset

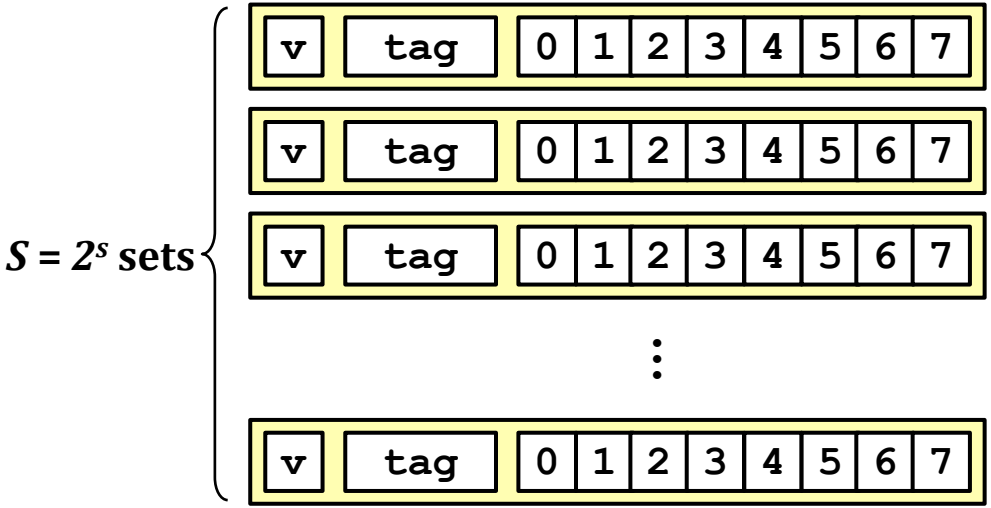
Address of Word:

t bits	s bits	b bits
Tag	Set Index	Block Offset



Example: Direct Mapped Cache (E = 1)

- Direct mapped: one line per set
  - Assume: cache block size is 8 bytes

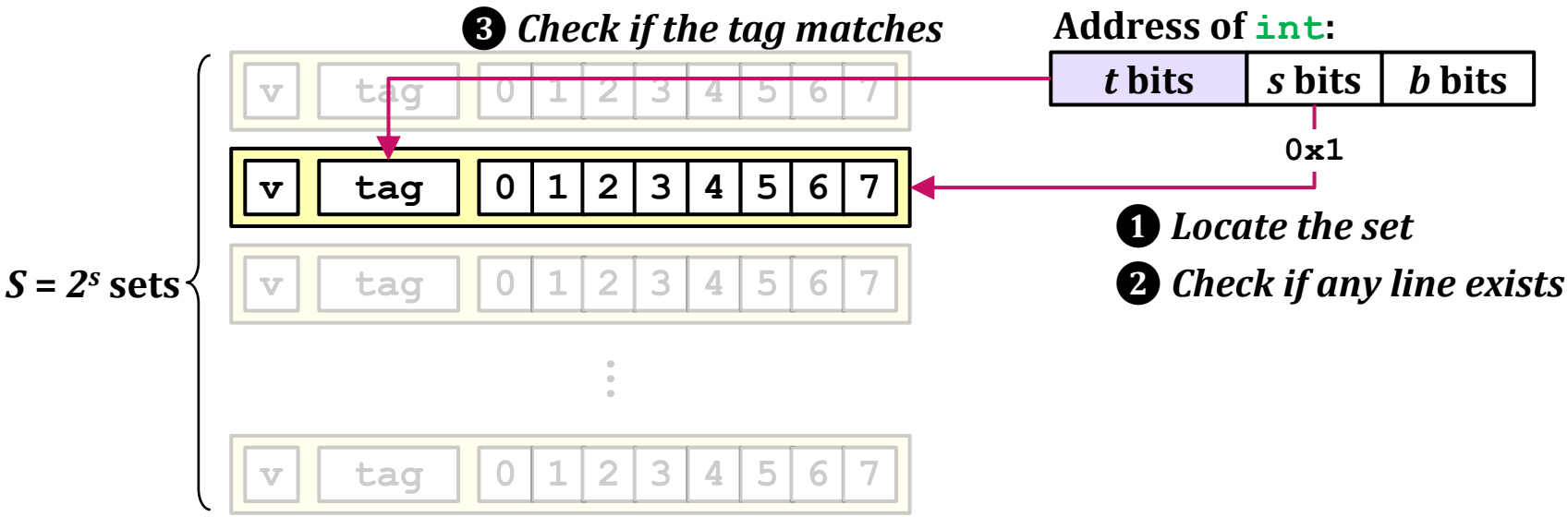


Address of `int`:

t bits	s bits	b bits

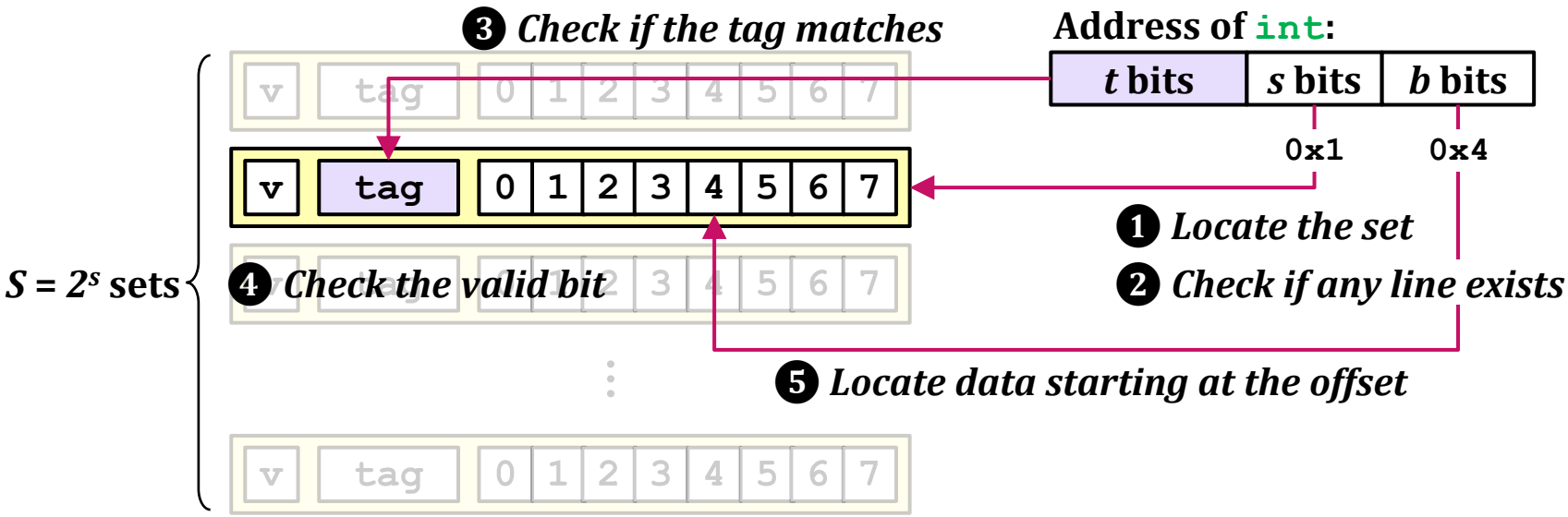
Example: Direct Mapped Cache (E = 1)

- Direct mapped: one line per set
  - Assume: cache block size is 8 bytes



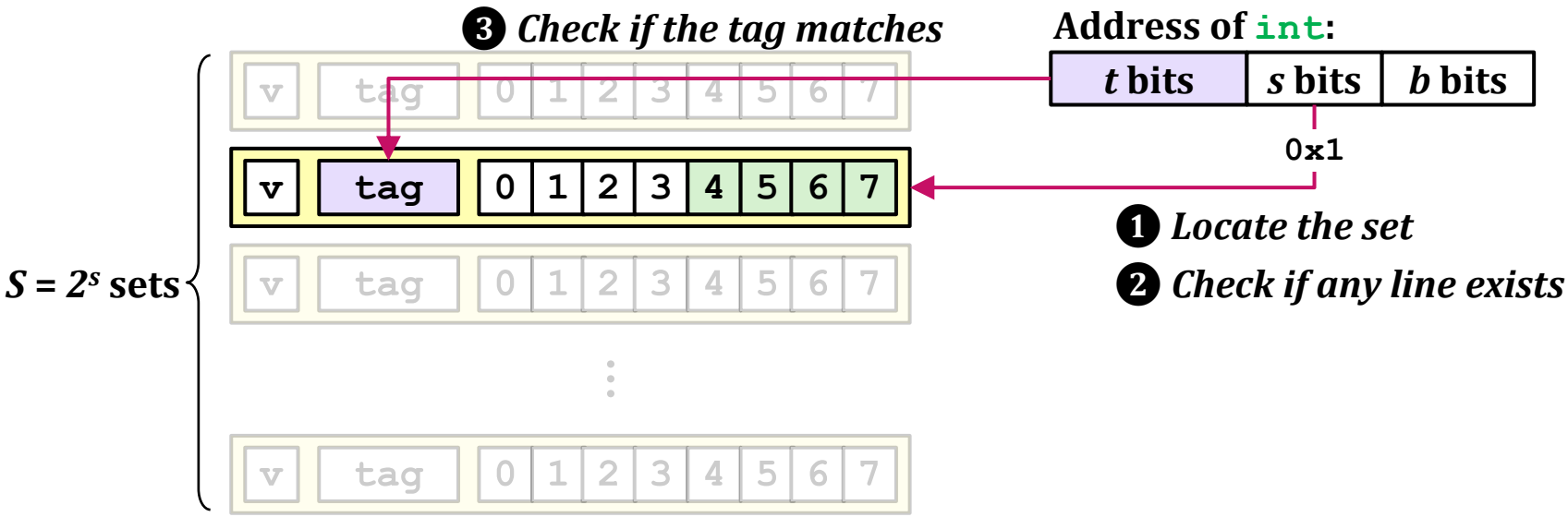
Example: Direct Mapped Cache (E = 1)

- Direct mapped: one line per set
  - Assume: cache block size is 8 bytes



Example: Direct Mapped Cache (E = 1)

- Direct mapped: one line per set
  - Assume: cache block size is 8 bytes



Upon miss (i.e., !(❷ && ❸ && ❹)): old line is evicted and replaced

Direct-Mapped Cache Simulation

- Configurations
  - M: 16-byte address space (4-bit addresses)
  - B: 2^1 bytes per block (b = 1: 1-bit block offset)
  - E: 2^0 blocks per set
  - S: 2^2 sets in the cache

t=1	s=2		b=1
t_0	s_1	s_0	b_0

Address Trace (1-Byte Reads):

0 [$0\underline{00}0_2$] **Miss**

	Valid	Tag	Block	
Set 0	–	–	–	–
Set 1	–	–	–	–
Set 2	–	–	–	–
Set 3	–	–	–	–

Direct-Mapped Cache Simulation

- Configurations
  - M: 16-byte address space (4-bit addresses)
  - B: 2^1 bytes per block (b = 1: 1-bit block offset)
  - E: 2^0 blocks per set
  - S: 2^2 sets in the cache

t=1	s=2		b=1
t_0	s_1	s_0	b_0

Address Trace (1-Byte Reads):

0	$[0\underline{00}0_2]$	Miss
1	$[0\underline{00}1_2]$	Hit
7	$[0\underline{11}1_2]$	Miss

	Valid	Tag	Block	
Set 0	1	0	M[0]	M[1]
Set 1	–	–	–	–
Set 2	–	–	–	–
Set 3	–	–	–	–

Direct-Mapped Cache Simulation

- Configurations
  - M: 16-byte address space (4-bit addresses)
  - B: 2^1 bytes per block (b = 1: 1-bit block offset)
  - E: 2^0 blocks per set
  - S: 2^2 sets in the cache

t=1	s=2		b=1
t_0	s_1	s_0	b_0

Address Trace (1-Byte Reads):

0	$[0\underline{00}0_2]$	Miss
1	$[0\underline{00}1_2]$	Hit
7	$[0\underline{11}1_2]$	Miss
8	$[1\underline{00}0_2]$	Miss

	Valid	Tag	Block	
Set 0	1	0	M[0]	M[1]
Set 1	–	–	–	–
Set 2	–	–	–	–
Set 3	1	0	M[6]	M[7]

Direct-Mapped Cache Simulation

- Configurations
  - M: 16-byte address space (4-bit addresses)
  - B: 2^1 bytes per block (b = 1: 1-bit block offset)
  - E: 2^0 blocks per set
  - S: 2^2 sets in the cache

t=1	s=2		b=1
t_0	s_1	s_0	b_0

Address Trace (1-Byte Reads):

0	$[0\underline{00}0_2]$	Miss
1	$[0\underline{00}1_2]$	Hit
7	$[0\underline{11}1_2]$	Miss
8	$[1\underline{00}0_2]$	Miss
0	$[0\underline{00}0_2]$	Miss

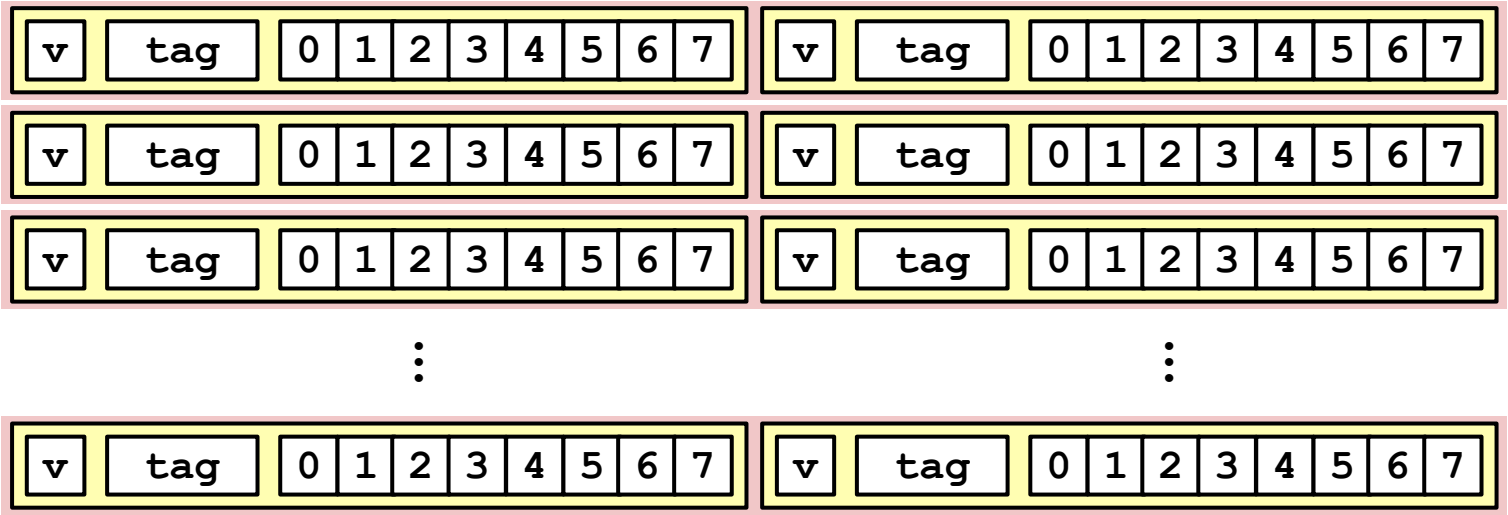
	Valid	Tag	Block	
Set 0	1	0	M[8]	M[9]
Set 1	–	–	–	–
Set 2	–	–	–	–
Set 3	1	0	M[6]	M[7]

E-Way Set-Associative Cache (e.g., E = 2)

- E = 2 lines per set
  - Assume: cache block size is 8 bytes

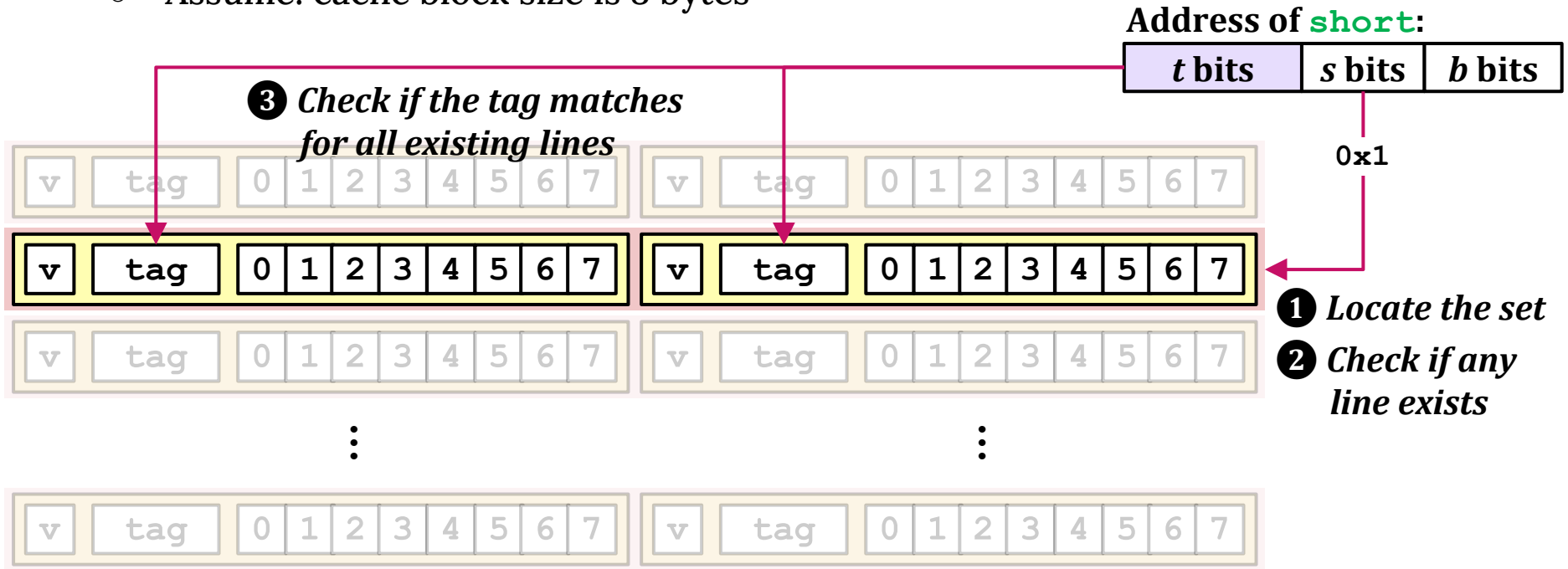
Address of `short`:

t bits	*s* bits	*b* bits



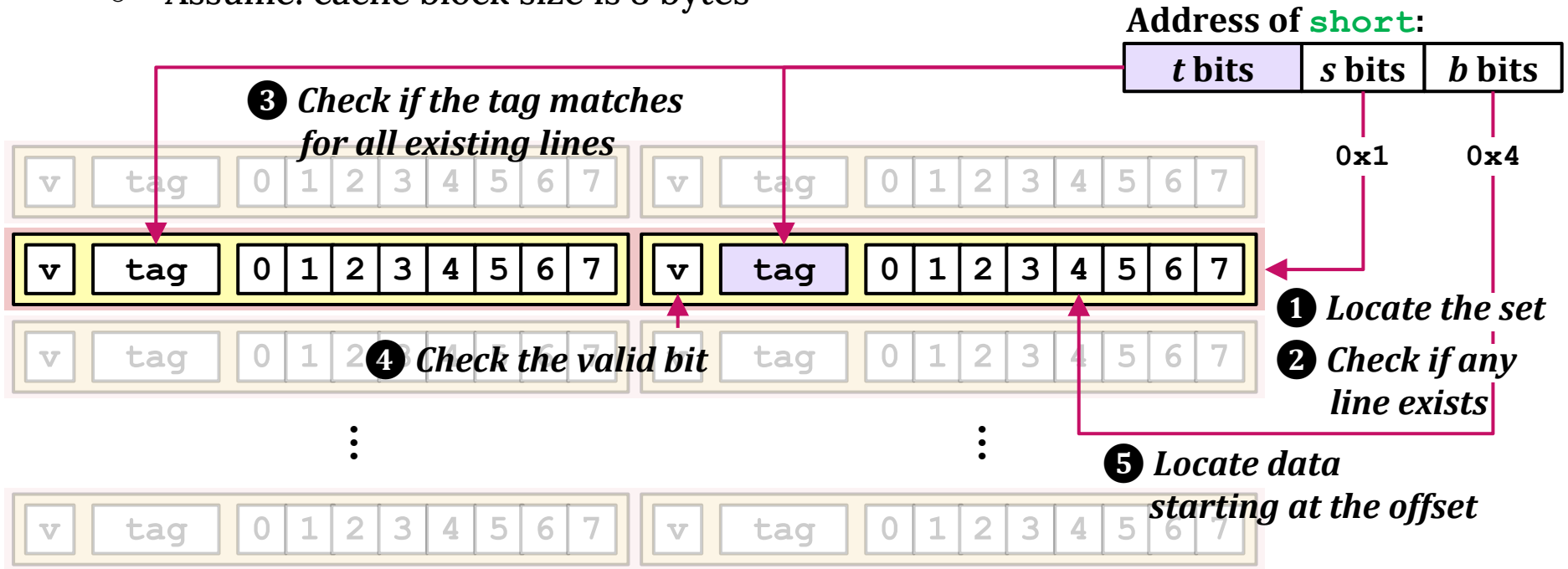
E-Way Set-Associative Cache (e.g., E = 2)

- E = 2 lines per set
  - Assume: cache block size is 8 bytes



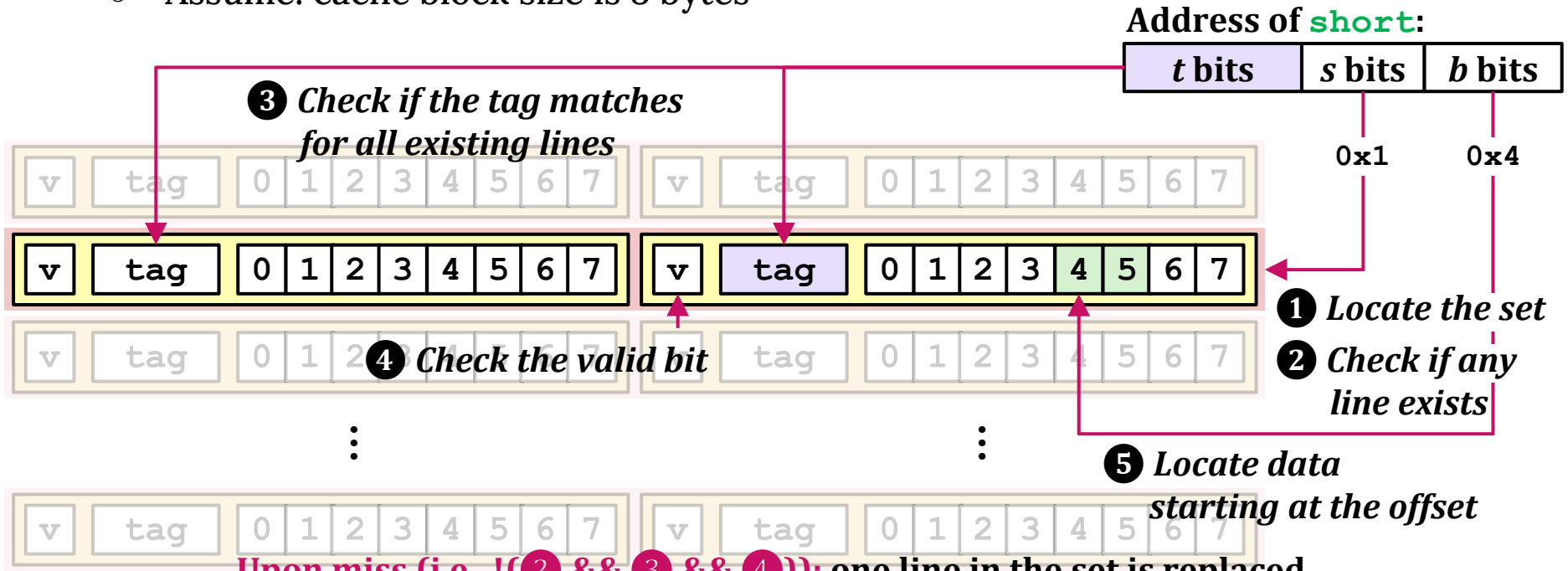
E-Way Set-Associative Cache (e.g., E = 2)

- E = 2 lines per set
  - Assume: cache block size is 8 bytes



E-Way Set-Associative Cache (e.g., E = 2)

- E = 2 lines per set
  - Assume: cache block size is 8 bytes

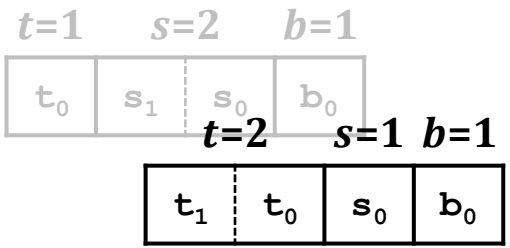


Upon miss (i.e., !(2 && 3 && 4)): one line in the set is replaced

Replacement policy: random, least-recently used (LRU), etc.

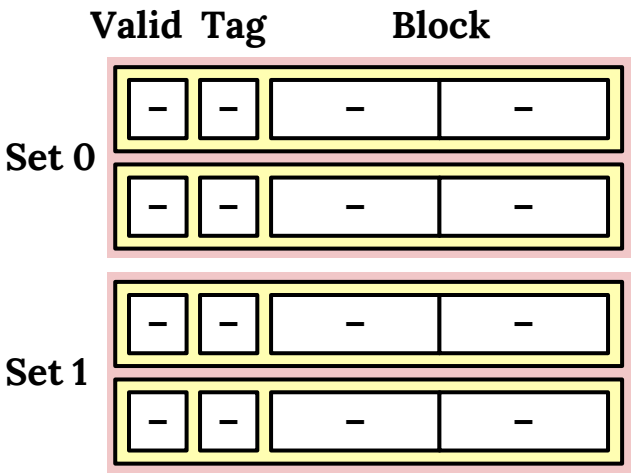
2-Way Set-Associative Cache Simulation

- Configurations
  - M: 16-byte address space (4-bit addresses)
  - B: 2^1 bytes per block (b = 1: 1-bit block offset)
  - E: ~~2^0~~ 2^1 blocks per set
  - S: ~~2^2~~ 2^1 sets in the cache



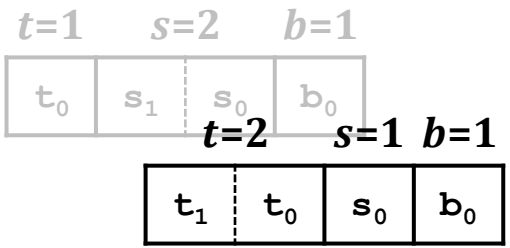
Address Trace (1-Byte Reads):

0 [$00\underline{0}0_2$] **Miss**



2-Way Set-Associative Cache Simulation

- Configurations
  - M: 16-byte address space (4-bit addresses)
  - B: 2^1 bytes per block (b = 1: 1-bit block offset)
  - E: ~~2^0~~ 2^1 blocks per set
  - S: ~~2^2~~ 2^1 sets in the cache



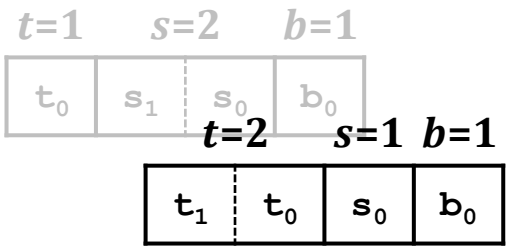
Address Trace (1-Byte Reads):

0	$[000\underline{0}_2]$	Miss
1	$[000\underline{1}_2]$	Hit
7	$[01\underline{1}1_2]$	Miss

	Valid	Tag	Block	
Set 0	1	00	M[0]	M[1]
	–	–	–	–
Set 1	–	–	–	–
	–	–	–	–

2-Way Set-Associative Cache Simulation

- Configurations
  - M: 16-byte address space (4-bit addresses)
  - B: 2^1 bytes per block (b = 1: 1-bit block offset)
  - E: ~~2^0~~ 2^1 blocks per set
  - S: ~~2^2~~ 2^1 sets in the cache



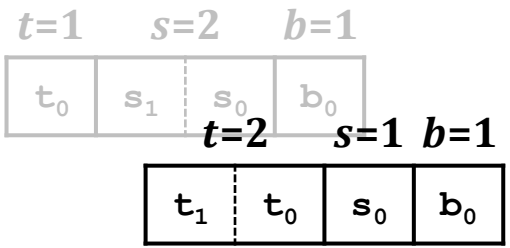
Address Trace (1-Byte Reads):

0	[000$_2$]	Miss
1	[001$_2$]	Hit
7	[011$_2$]	Miss
8	[100$_2$]	Miss

	Valid	Tag	Block	
Set 0	1	00	M[0]	M[1]
	–	–	–	–
Set 1	1	01	M[6]	M[7]
	–	–	–	–

2-Way Set-Associative Cache Simulation

- Configurations
  - M: 16-byte address space (4-bit addresses)
  - B: 2^1 bytes per block (b = 1: 1-bit block offset)
  - E: ~~2^0~~ 2^1 blocks per set
  - S: ~~2^2~~ 2^1 sets in the cache



Address Trace (1-Byte Reads):

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

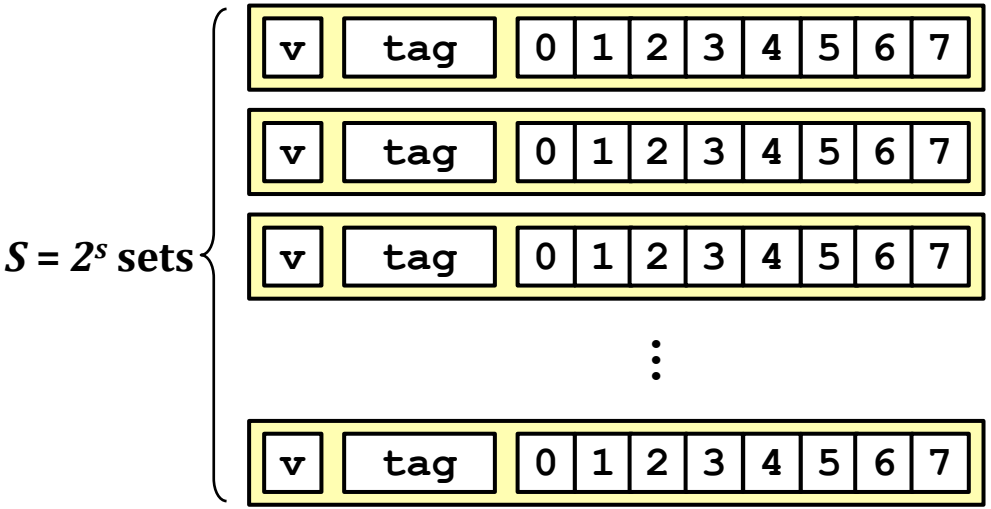
	Valid	Tag	Block	
Set 0	1	00	M[0]	M[1]
	1	10	M[8]	M[9]
Set 1	1	01	M[6]	M[7]
	–	–	–	–

What About Writes?

- Multiple copies of data exist across memory hierarchy
  - L1, L2, L3, main memory, and disk
- Write-hit handling
  - Write-through: write the data immediately to memory
  - Write-back: defer write to memory until replacement of line
    - Need a dirty bit to mark whether the line is different from memory or not
- Write-miss handling
  - Write-allocate: load into cache and update line in cache
    - Good if more writes to the location follow
  - No-write-allocate: write immediately to memory
- Typically: [write-through + no-write-allocate] or [write-back + write-allocate]

Why Index Using Middle Bits?

- Direct mapped: one line per set
  - Assume: cache block size is 8 bytes



Standard Method:
Middle-Bit Indexing

Address of `int`:

t bits	*s* bits	*b* bits

Find set

Alternative Method:
High-Bit Indexing

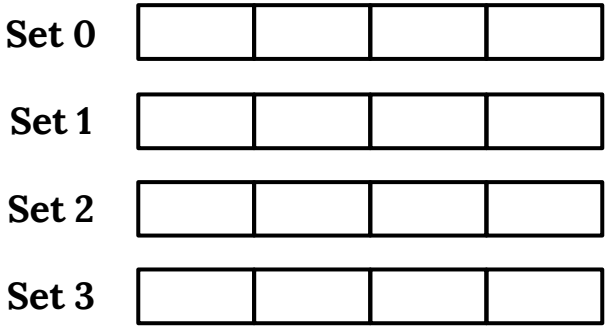
Address of `int`:

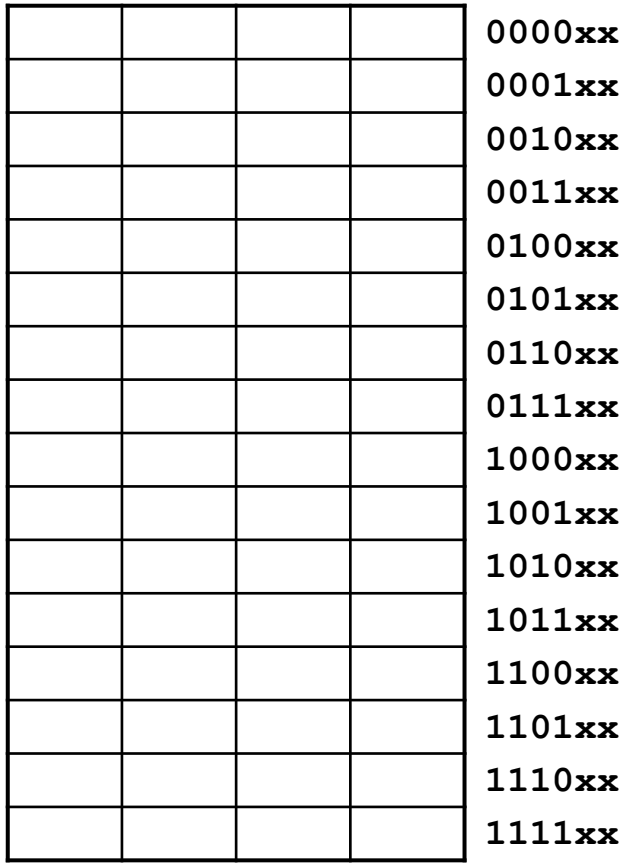
s bits	*t* bits	*b* bits

Find set

Illustration of Indexing Approaches

- 64-byte memory → 6-bit addresses
- 16-byte direct-mapped cache
  - Block size: 4 bytes
  - Direct mapped: one-block per set → 4 sets
- 2-bit block offset, 2-bit set index → 2-bit tag

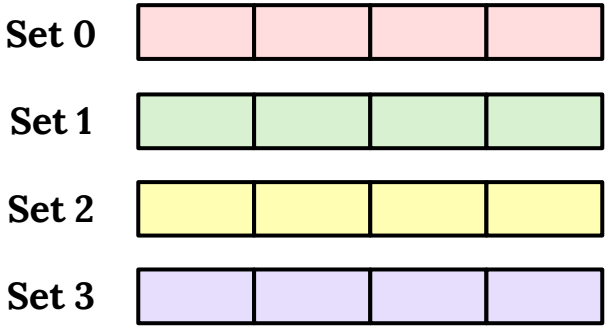


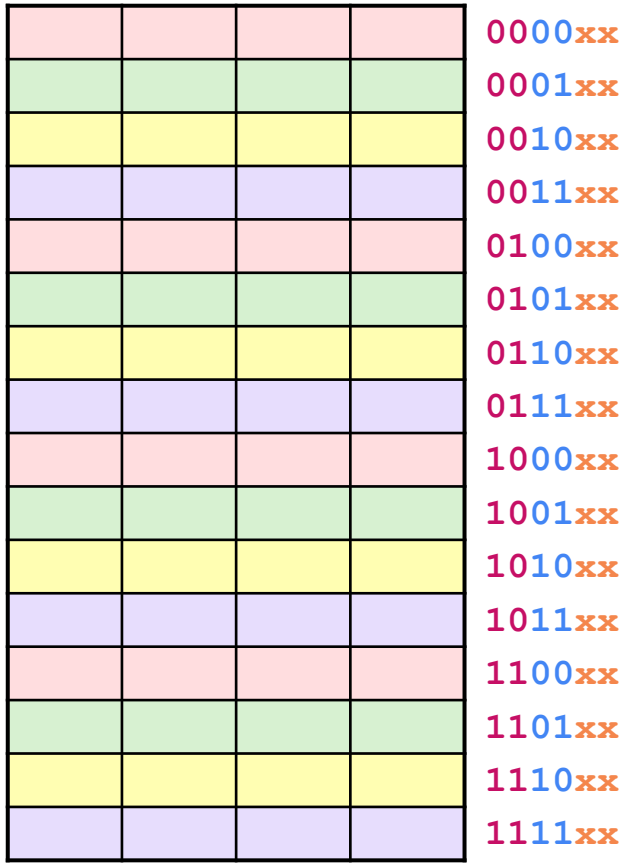


Middle-Bit Indexing

- Addresses of form `ttssbb`
  - `tt`: tag bits
  - `ss`: set index bits
  - `bb`: offset bits

- Makes good use of spatial locality

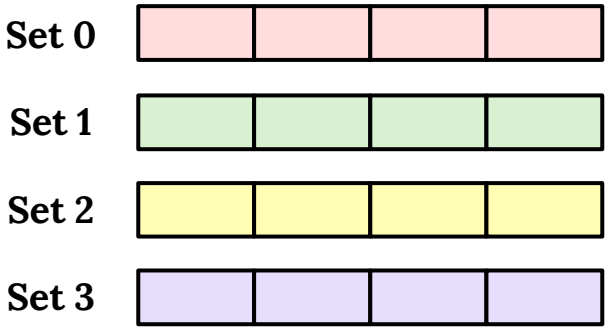


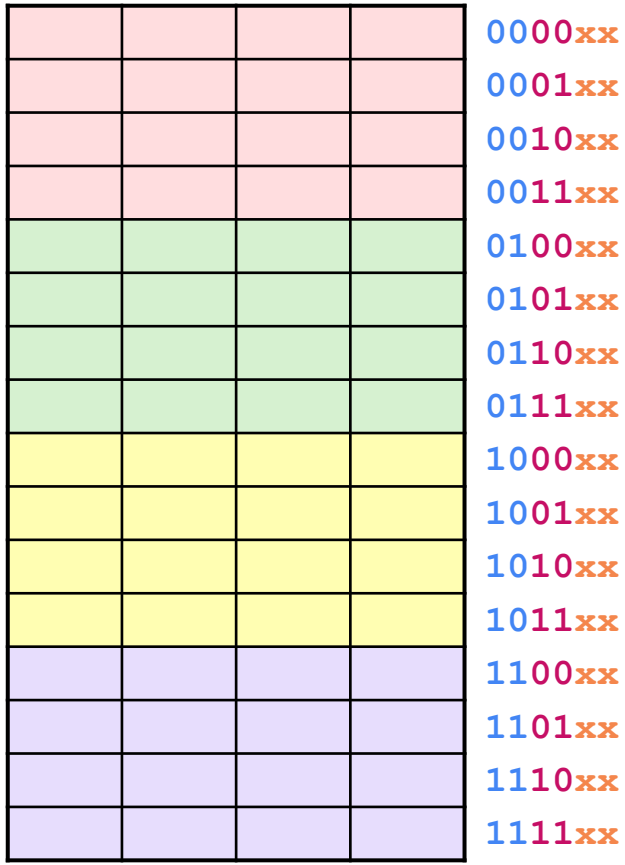


High-Bit Indexing

- Addresses of form `ssttbb`
  - `tt`: tag bits
  - `ss`: set index bits
  - `bb`: offset bits

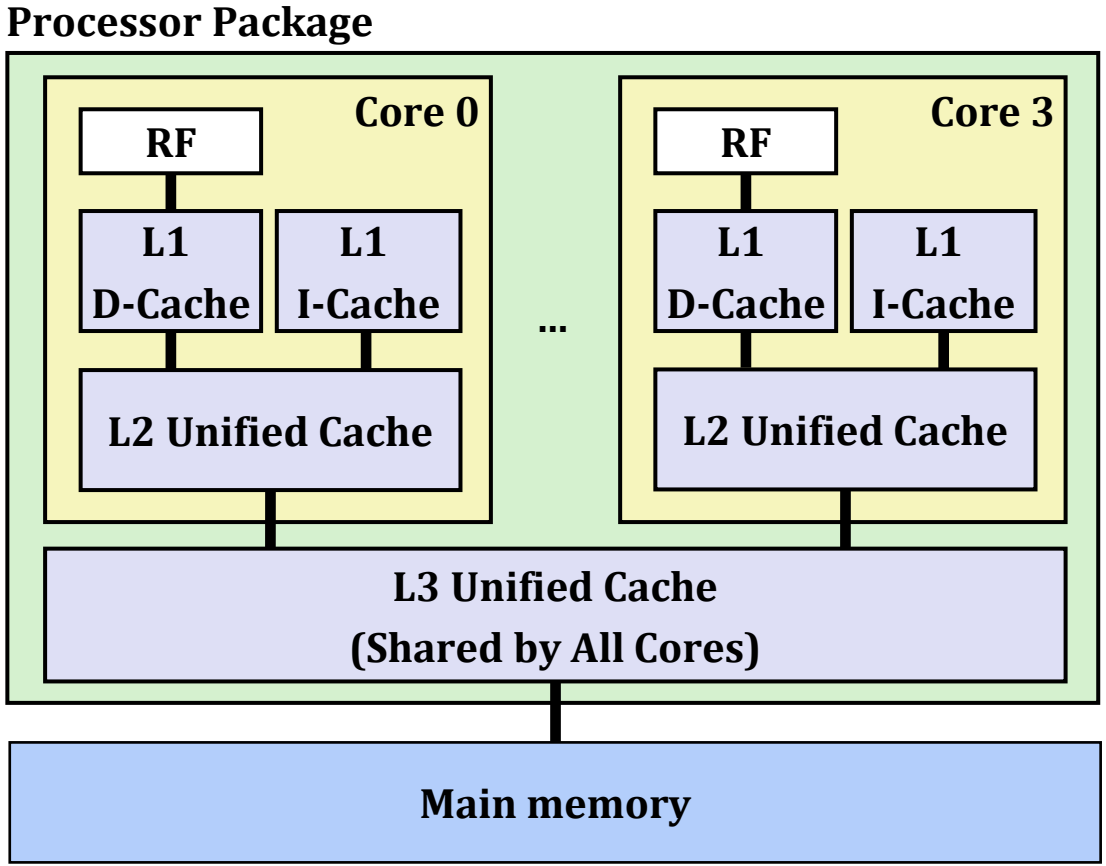
- Program with high spatial locality would generate lots of conflicts





Intel Core i7 Cache Hierarchy

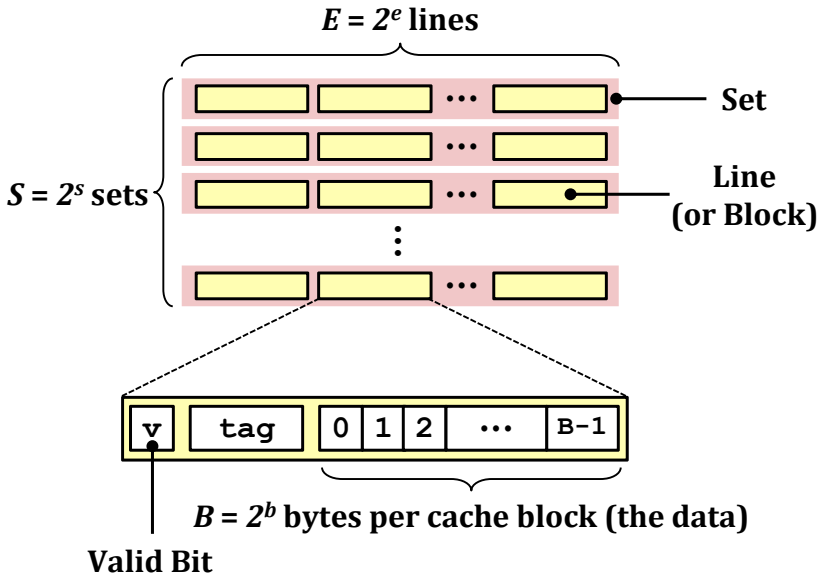
- L1 i-cache and d-cache
  - 32 KB, 8 ways
  - Access latency: 4 cycles
- L2 unified cache
  - 256 KB, 8 ways
  - Access latency: 11 cycles
- L3 unified cache
  - 8 MB, 16 ways
  - Acc. lat.: 30 – 40 cycles
- Block size
  - 64 bytes for all caches



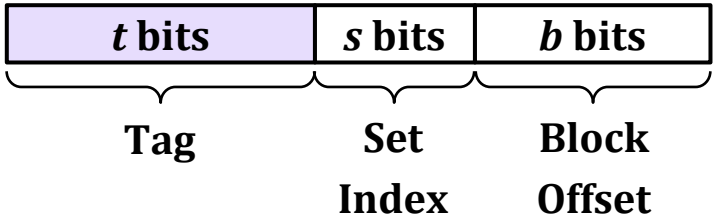
Example: Intel Core i7 L1 Data Cache

- Configurations
  - 32-KB 8-way set associative
  - 64 bytes per block
  - 47-bit address range

B = 64 (b = 6)
S = 64 (s = 6)
E = 8 (e = 3)
C = 32,768 = 2^{15}



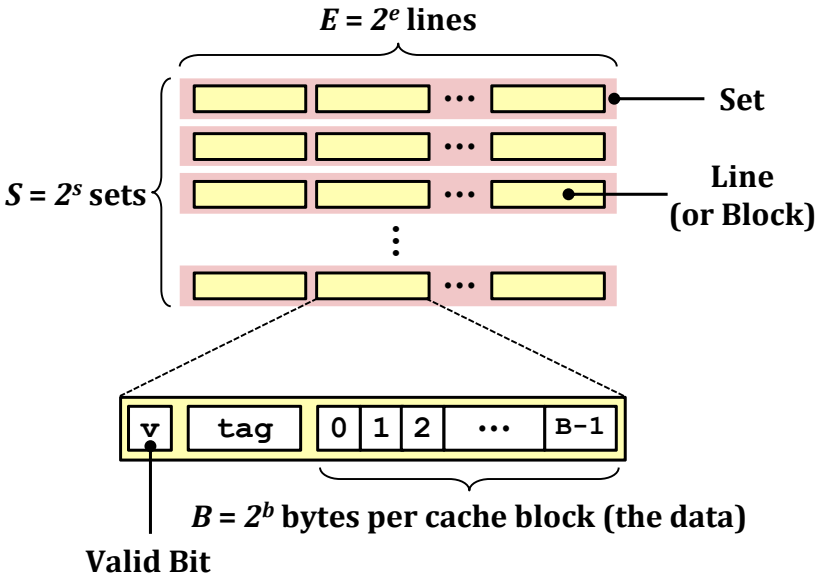
Address of Word:



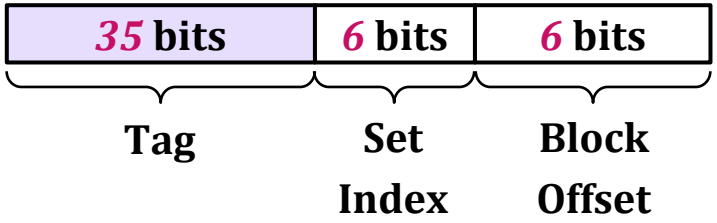
Example: Intel Core i7 L1 Data Cache

- Configurations
  - 32-KB 8-way set associative
  - 64 bytes per block
  - 47-bit address range

B = 64 (b = 6)
S = 64 (s = 6)
E = 8 (e = 3)
C = 32,768 = 2^{15}



Address of Word:



Stack Address: `0x00007f7262a1e010`

Binary

Tag

`0000 0001 0000`

Set Index: `0x0`

Block Offset: `0x10`

Cache Perfomance Metrics

- Miss rate: fraction of memory references not found in cache
  - 1 – (*hit rate*)
  - Typical numbers (in percentages)
    - 3-10% for L1
    - Can be quite small (e.g., < 1%) for L2, depending on size, etc.
- Hit time: time to deliver a line in the cache to the processor
  - Includes time to determine whether the line is in the cache
  - Typical numbers
    - 4 clock cycles for L1
    - 10 clock cycles for L2
- Miss penalty: additional time required because of a miss
    - Typically 50-200 cycles for main memory (trend: increasing)

Let's Think About Those Numbers

- Huge difference between the hit time and the miss penalty
  - Could be 100×, if just L1 and main memory

- High performance impact of hit/miss rate
  - e.g., 99% hit rate → 97% hit rate
  - Assumptions
    - Hit time: 1 cycle
    - Miss penalty: 100 cycles
  - Average access time:
    - 97% hits: 0.97×1 cycle + 0.03×100 cycles = **3.97 cycles**
    - 99% hits: 0.99×1 cycle + 0.01×100 cycles = **1.99 cycles**
  - **99% hit rate is twice as good as 97%**
    → This is why miss rate is used instead of hit rate

Writing Cache Friendly Code

- Make the common case go fast
  - Focus on the inner loops of the core functions

- Minimize the misses in the inner loops
  - Repeated references to variables are good (temporal locality)
  - Stride-1 reference patterns are good (spatial locality)

- Key idea: our qualitative notion of locality is quantified through our understanding of cache memory

Lecture Agenda

- Cache memory organization and operation
- Performance impact of caches
  - The memory mountain
  - Rearranging loops to improve spatial locality
  - Using blocking to improve temporal locality

The Memory Mountain

- Read throughput (read bandwidth)
  - Number of bytes read from memory per second (MB/s)

- Memory mountain
  - Measured read throughput as a function of spatial and temporal locality
    - Compact way to characterize memory system performance.

Memory Mountain Test Function

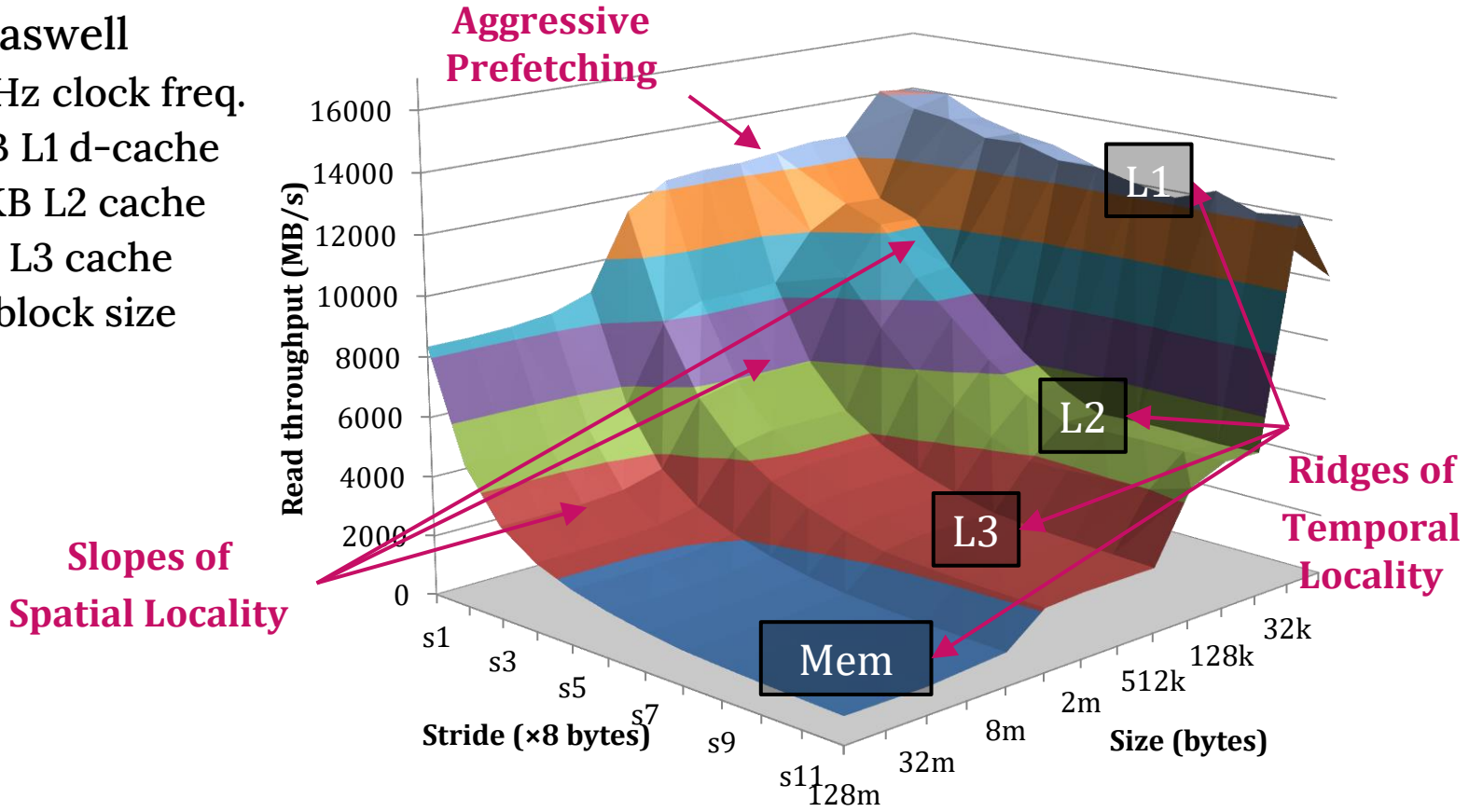
- Call `test()` with many combiations of # of elements and stride

- For each combination
  - Call `test()` once to warm up the caches
  - Call `test()` again and measure the read throughput (MB/s)

```
long data[MAXELEMS];  // Global array to traverse
// test - Iterate over first "elems" elements of
//        array "data" with stride of "stride",
//        using 4x4 loop unrolling.
int test(int elems, int stride){
    long i, sx2=stride*2, sx3=stride*3, sx4=stride*4;
    long acc0 = 0, acc1 = 0, acc2 = 0, acc3 = 0;
    long length = elems, limit = length - sx4;
    // Combine 4 elements at a time
    for (i = 0; i < limit; i += sx4) {
        acc0 = acc0 + data[i];
        acc1 = acc1 + data[i + stride];
        acc2 = acc2 + data[i + sx2];
        acc3 = acc3 + data[i + sx3];
    }
    // Finish any remaining elements
    for (; i < length; i++)
        acc0 = acc0 + data[i];
    return ((acc0 + acc1) + (acc2 + acc3));
}
```

The Memory Mountain

- Core i7 Haswell
  - 2.1-GHz clock freq.
  - 32-KB L1 d-cache
  - 256-KB L2 cache
  - 8-MB L3 cache
  - 64-B block size



Lecture Agenda

- Cache memory organization and operation
- Performance impact of caches
  - The memory mountain
  - Rearranging loops to improve spatial locality
  - Using blocking to improve temporal locality

Matrix Multiplication Example

- Description
  - Multiply N×N matrices
  - $O(N^3)$ total operations
  - N reads per source elements
  - N values summed per destination
    - But may be able to hold in register

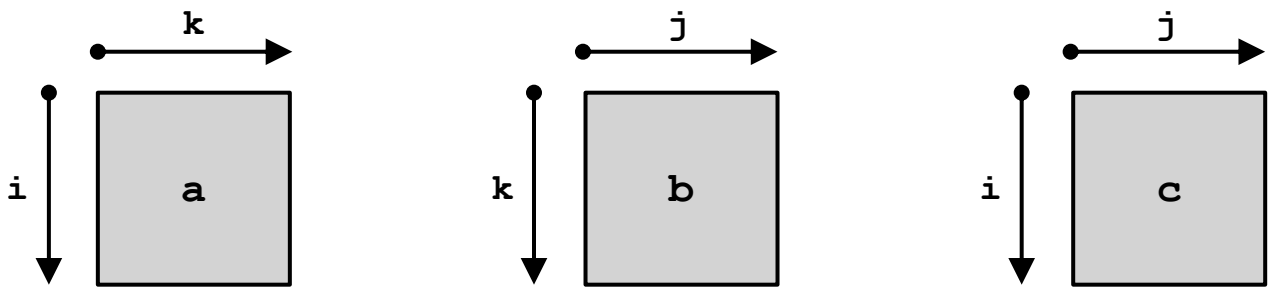
```
/* ijk */
for (i = 0; i < n; i++){
  for (j = 0; j < n; j++){
    sum = 0.0;
    for (k = 0; k < n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

Variable `sum` held in register

Miss Rate Analysis for Matrix Multiply

- Assumptions
  - Line size: 32 bytes (big enough for four 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 the access pattern of inner loop



Layout of C Arrays in Memory (Review)

- In C language, an array is allocated in row-major order
  - each row in contiguous memory locations
- Stepping through columns in one row

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

  - Accesses successive elements
  - If block size B > 4 bytes, exploit spatial locality
    - Compulsory miss rate = 4 bytes / B
- Stepping through rows in one column

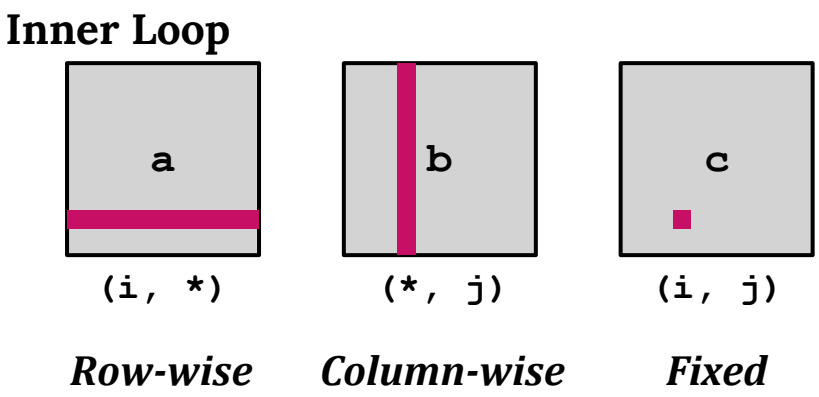
```
for (i = 0; i < n; i++)
    sum += a[i][0];
```

  - Accesses distant elements
  - No spatial locality
    - Compulsory miss rate = 1 (i.e., 100%)

Matrix Multiplication (`ijk`)

- Misses per inner loop iteration
  - `a`: 0.25
  - `b`: 1.0
  - `c`: 0.0

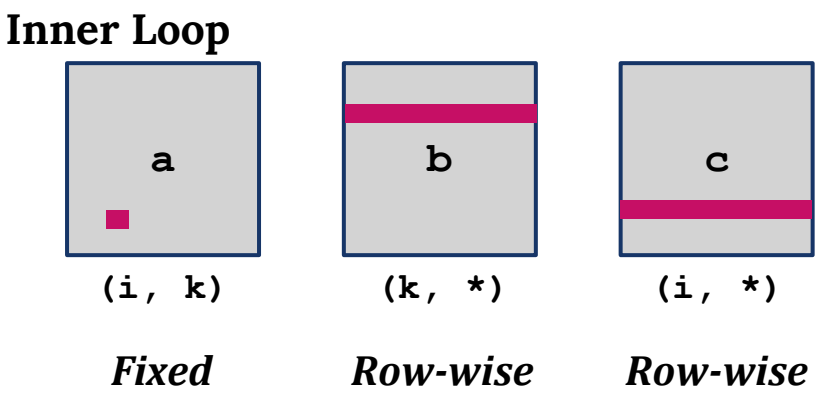
```
/* ijk */
for (i = 0; i < n; i++){
  for (j = 0; j < n; j++){
    sum = 0.0;
    for (k = 0; k < n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```



Matrix Multiplication (`kij`)

- Misses per inner loop iteration
  - `a`: 0.0
  - `b`: 0.25
  - `c`: 0.25

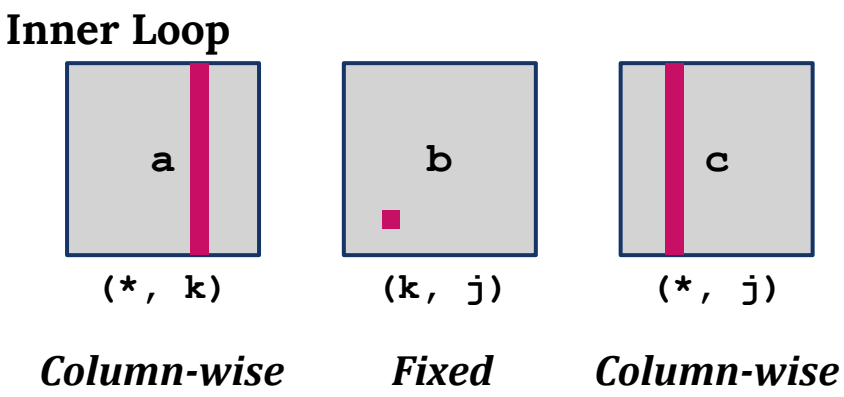
```
/* kij */
for (k = 0; k < n; k++){
  for (i = 0; i < n; i++){
    r = a[i][k];
    for (j = 0; j < n; j++)
      c[i][j] += r * b[k][j];
  }
}
```



Matrix Multiplication (`jki`)

- Misses per inner loop iteration
  - **a**: 1.0
  - **b**: 0.0
  - **c**: 1.0

```
/* 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;
  }
}
```



Summary of Matrix Multiplication

- **`ijk` (& `jik`)**
  - 2 loads and 0 store
  - misses per iteration = 1.25

```
for (i = 0; i < n; i++){      /* ijk */
  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;
  }
}
```

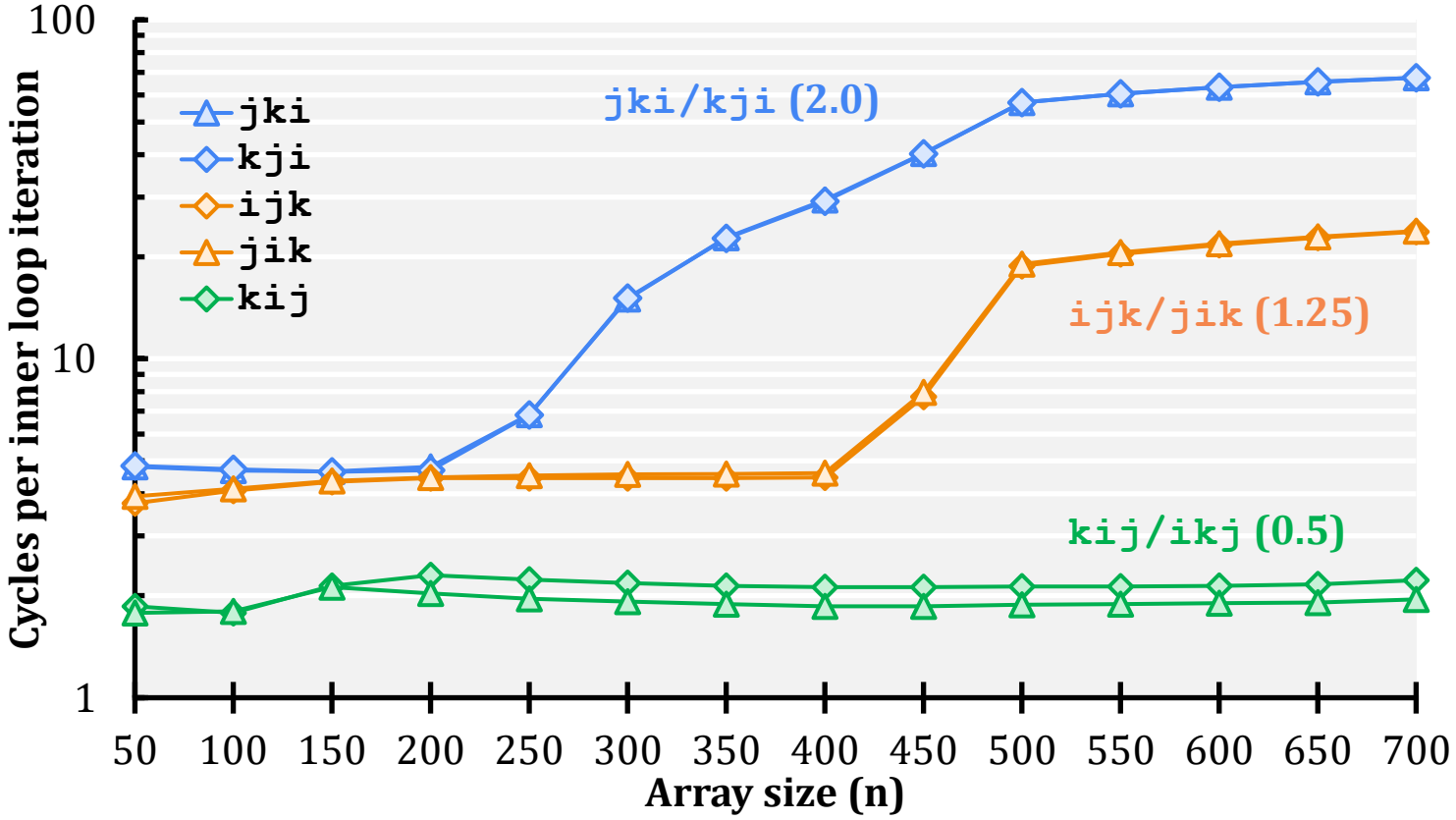
- **`kji` (& `ikj`)**
  - 2 loads, 1 store
  - misses per iteration = 0.5

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

- **`jki` (& `kji`)**
  - 2 loads and 1 store
  - misses per iteration = 2.0

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

Core i7 Matrix Multiply Performance



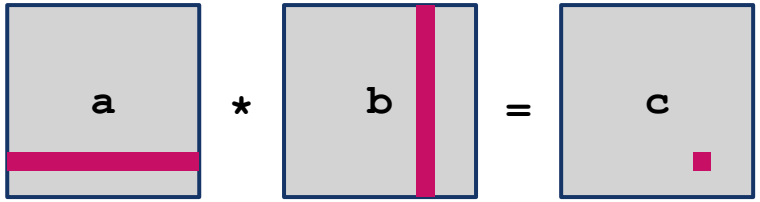
Lecture Agenda

- Cache memory organization and operation
- Performance impact of caches
  - The memory mountain
  - Rearranging loops to improve spatial locality
  - Using blocking to improve temporal locality

Example: Matrix Multiplication

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

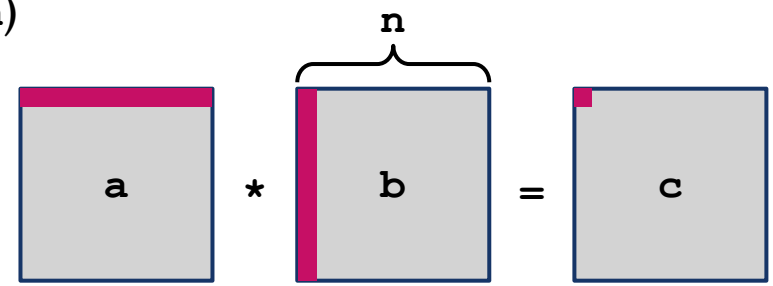


(i, *) (*, j) (i, j)

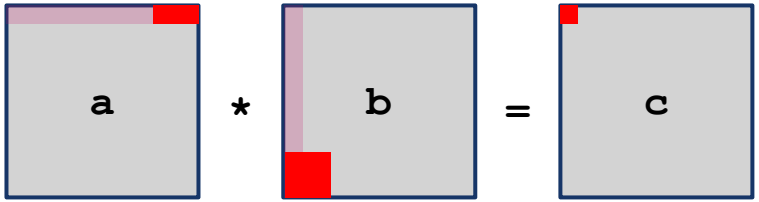
Cache Miss Analysis

- Assumptions
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size C << `n` (much smaller than `n`)

- First iteration
  - `n`/8 + `n` = 9`n`/8 misses



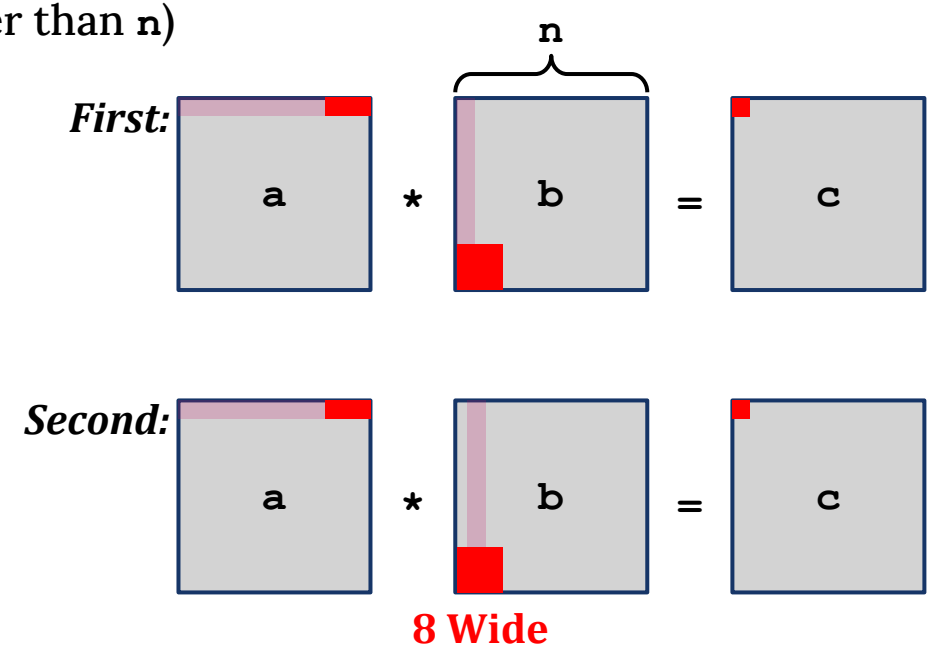
Afterwards in cache



8 Wide

Cache Miss Analysis

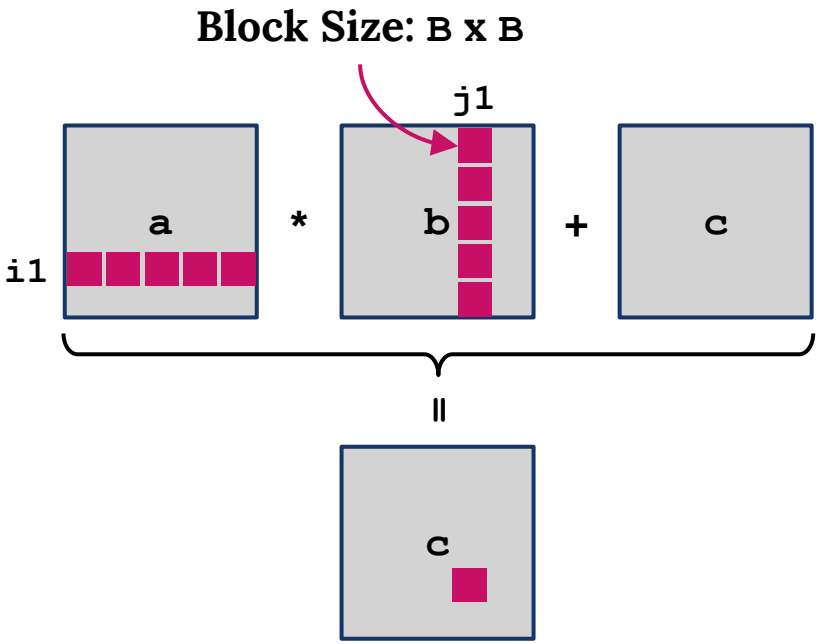
- Assumptions
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size C << **n** (much smaller than **n**)
- First iteration
  - **n**/8 + **n** = 9**n**/8 misses
- Second iteration
  - Same as the first iteration
- Total misses
  - $9\mathbf{n}/8 \times \mathbf{n}^2 = 9/8 \times \mathbf{n}^3$



Blocked Matrix Multiplication

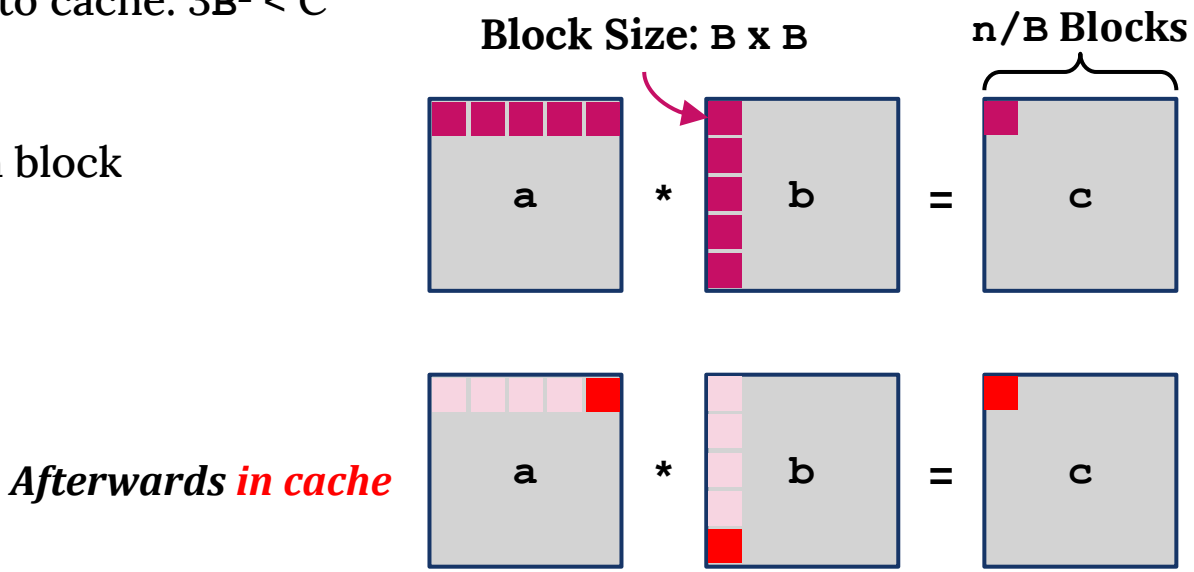
```
c = (double *) calloc(sizeof(double), n * n);

/* 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 += B)
      for (j = 0; j < n; j += B)
        for (k = 0; k < n; k += B)
/* B x B mini matrix multiplications */
          for (i1 = i; i1 < i + B; i++)
            for (j1 = j; j1 < j + B; j++)
              for (k1 = k; k1 < k + B; k++)
                c[i1 * n + j1] +=
                  a[i1 * n + k1] *
                  b[k1 * n + j1];
}
```



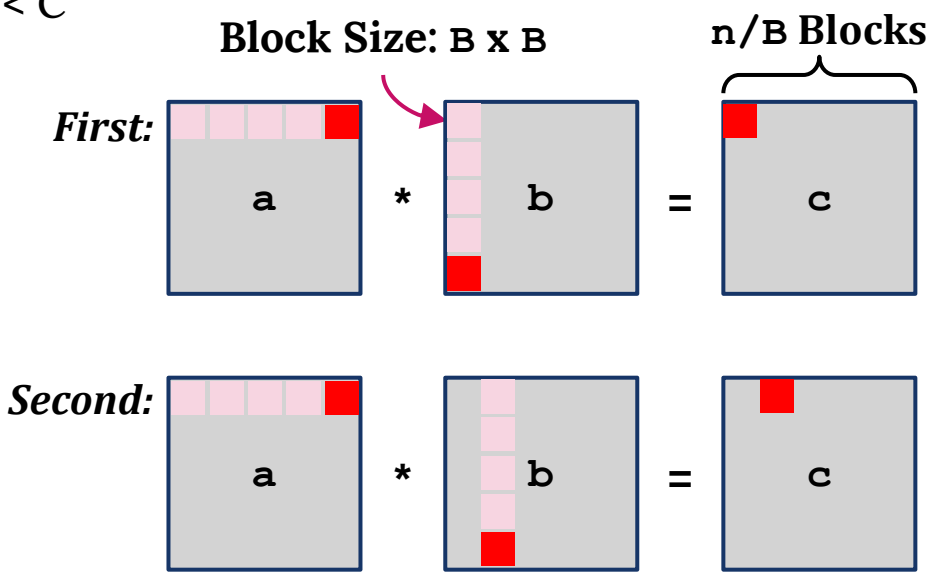
Cache Miss Analysis

- Assumptions
  - Cache block = 8 doubles
  - Cache size C << `n` (much smaller than `n`)
  - Three blocks ■ fit into cache: $3\mathtt{B}^2 < C$
- First (block) iteration
  - $\mathtt{B}^2/8$ misses for each block
  - $2\mathtt{n}/\mathtt{B} \times \mathtt{B}^2/8 = \mathtt{nB}/4$
    (omitting matrix c)



Cache Miss Analysis

- Assumptions
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)
  - Three blocks ■ fit into cache: $3B^2 < C$
- First (block) iteration
  - $B^2/8$ misses for each block
  - $2n/B \times B^2/8 = nB/4$
    (omitting matrix c)
- Second (block) iteration
  - Same as the first iteration
- Total misses
  - $nB/4 \times (n/B)^2 = n^3/4B$



Blocking Summary

- No blocking: $9/8 \times \mathbf{n}^3$
- Blocking: $(1/4\mathbf{B}) \times \mathbf{n}^3$
- Suggest largest possible block size **B**, but limit $3\mathbf{B}^2 < C$
- Reason for dramatic difference
  - Matrix multiplication has inherent temporal locality
    - Input data: $3\mathbf{n}^2$, computation: $2\mathbf{n}^3$
    - Every array elements used $O(\mathbf{n})$ times
  - But program has to be written properly

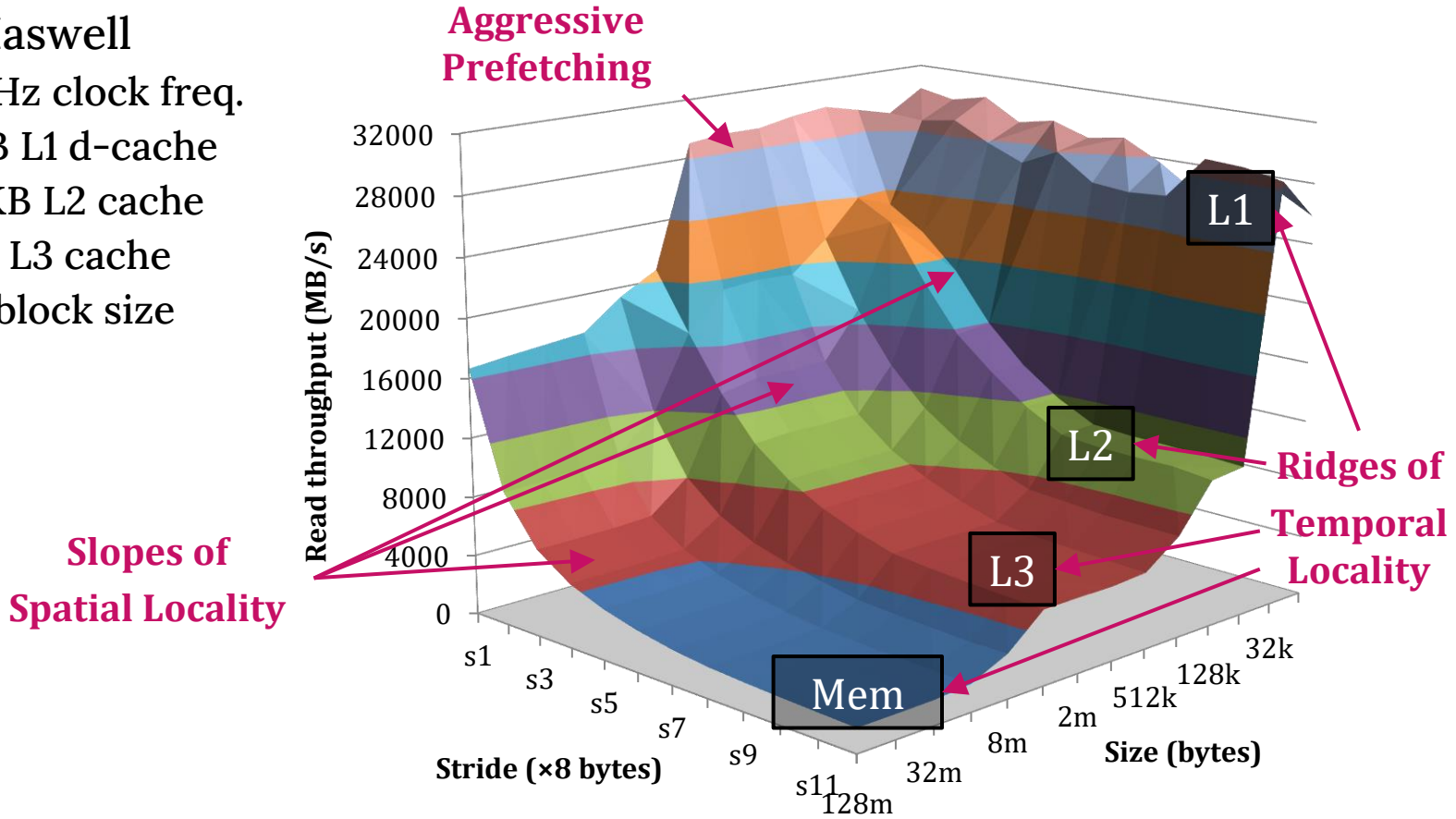
Cache Summary

- Cache can have significant performance impact

- You can (or had better to) write your programs to exploit this
  - Focus on the inner loops, where bulk computations and memory accesses occur
  - Try to maximize spatial locality by sequentially reading data objects with stride 1
  - Try to maximize temporal locality using a read data object as much as possible

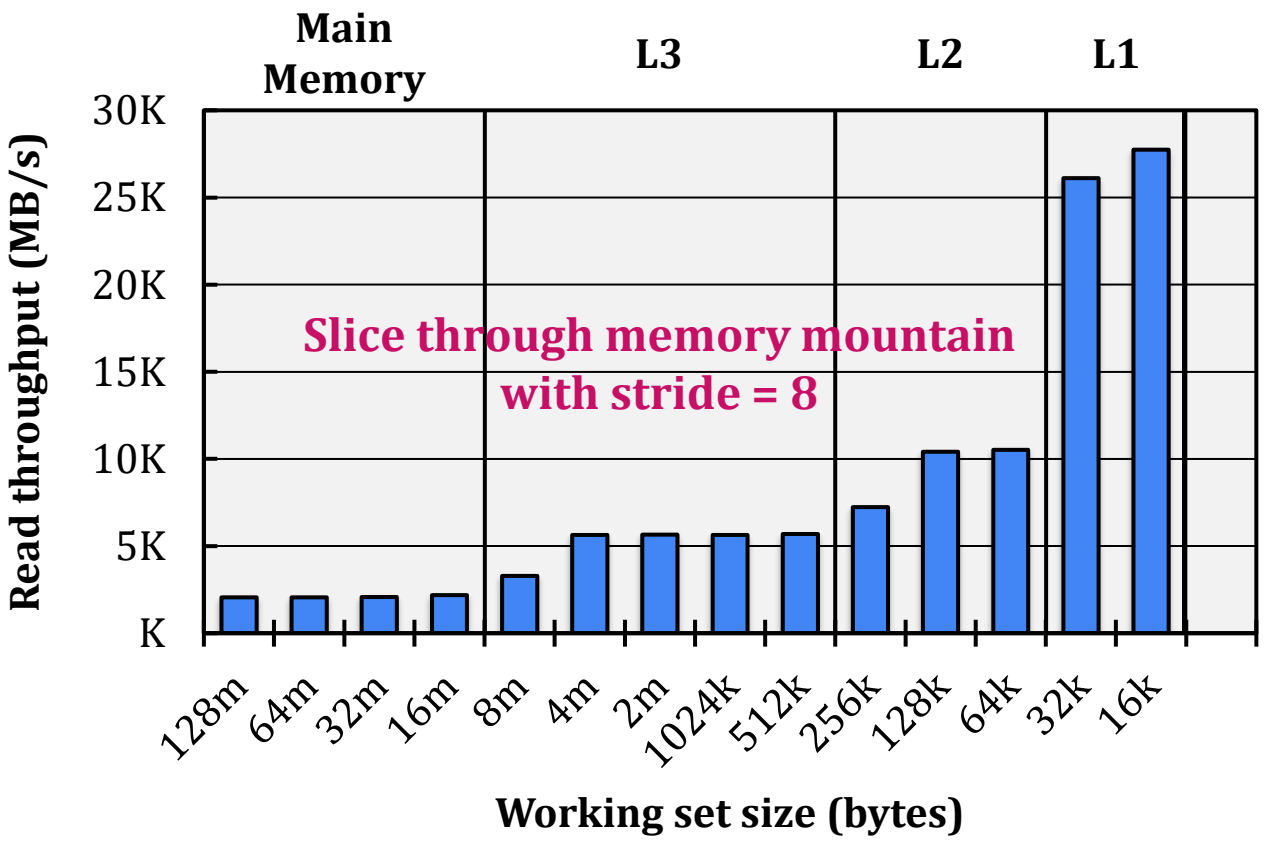
The Memory Mountain

- Core i5 Haswell
  - 3.1-GHz clock freq.
  - 32-KB L1 d-cache
  - 256-KB L2 cache
  - 8-MB L3 cache
  - 64-B block size



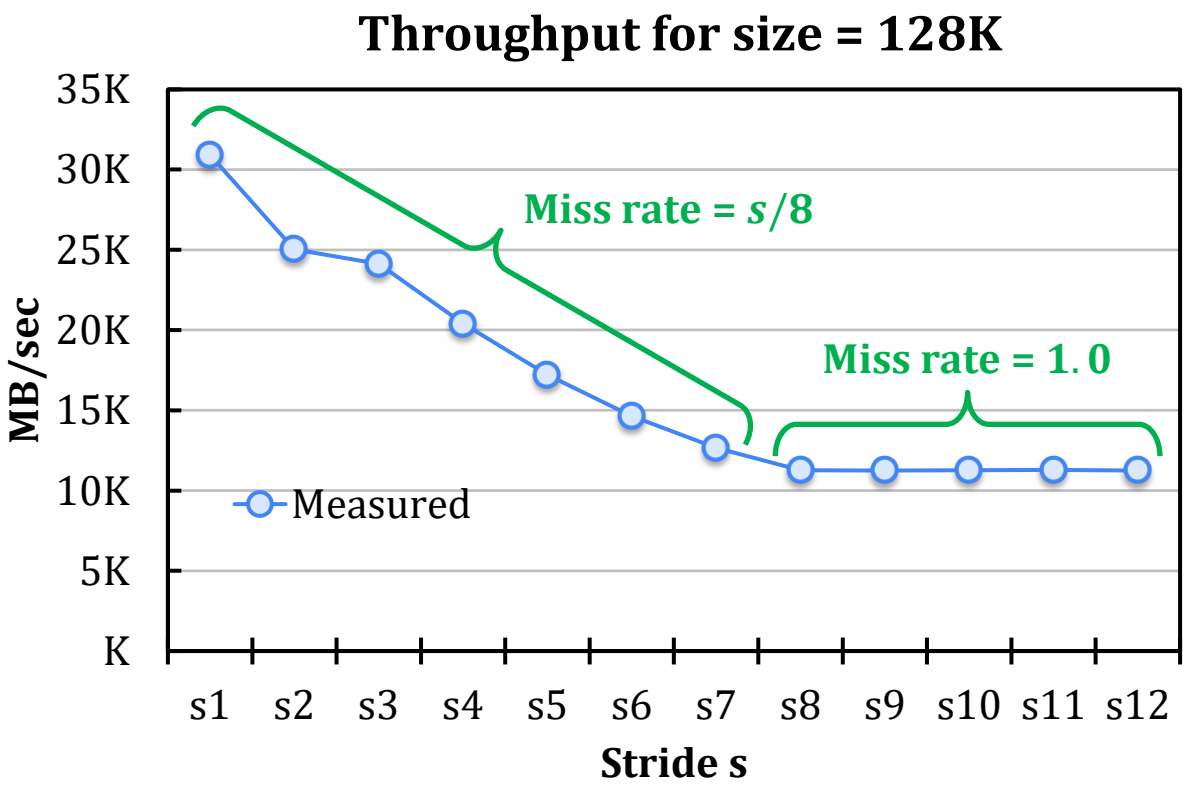
Cache Capcity Effects from Memory Mountain

- Core i7 Haswell
  - 2.1-GHz clock freq.
  - 32-KB L1 d-cache
  - 256-KB L2 cache
  - 8-MB L3 cache
  - 64-B block size



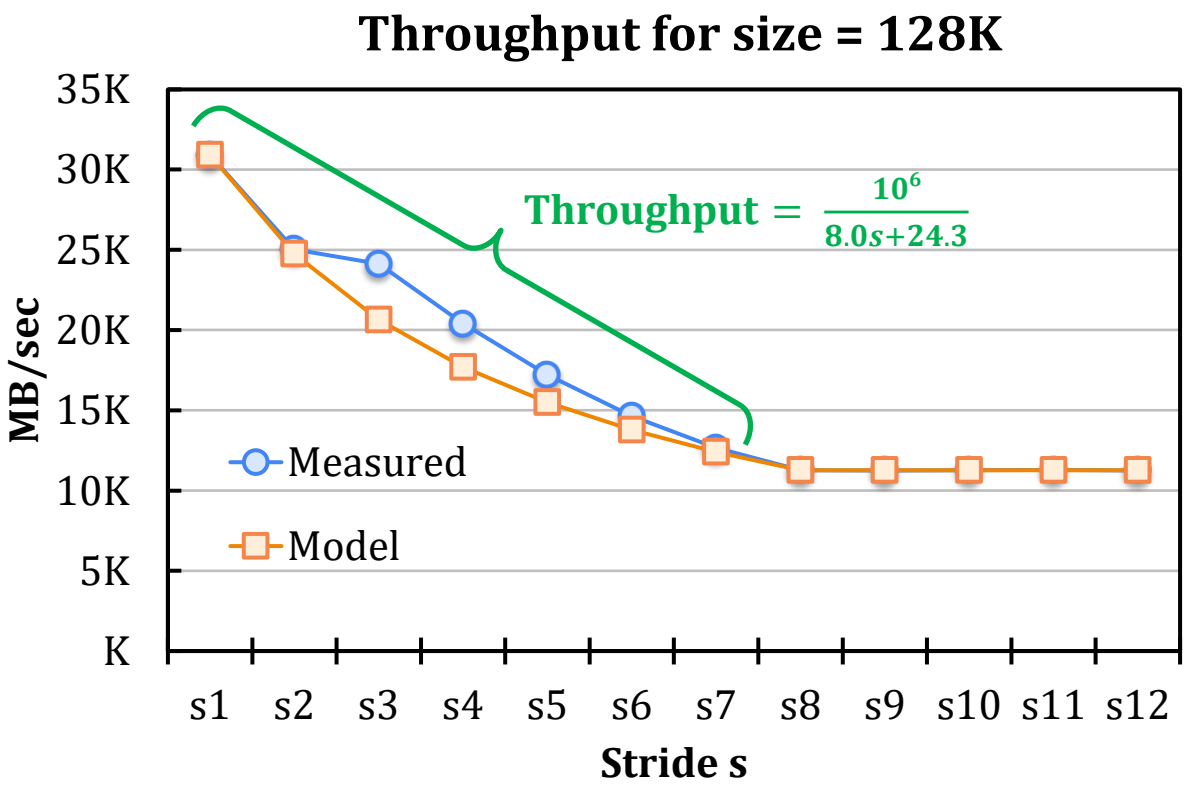
Cache Block Size Effects from Memory Mountain

- Core i7 Haswell
  - 2.1-GHz clock freq.
  - 32-KB L1 d-cache
  - 256-KB L2 cache
  - 8-MB L3 cache
  - 64-B block size



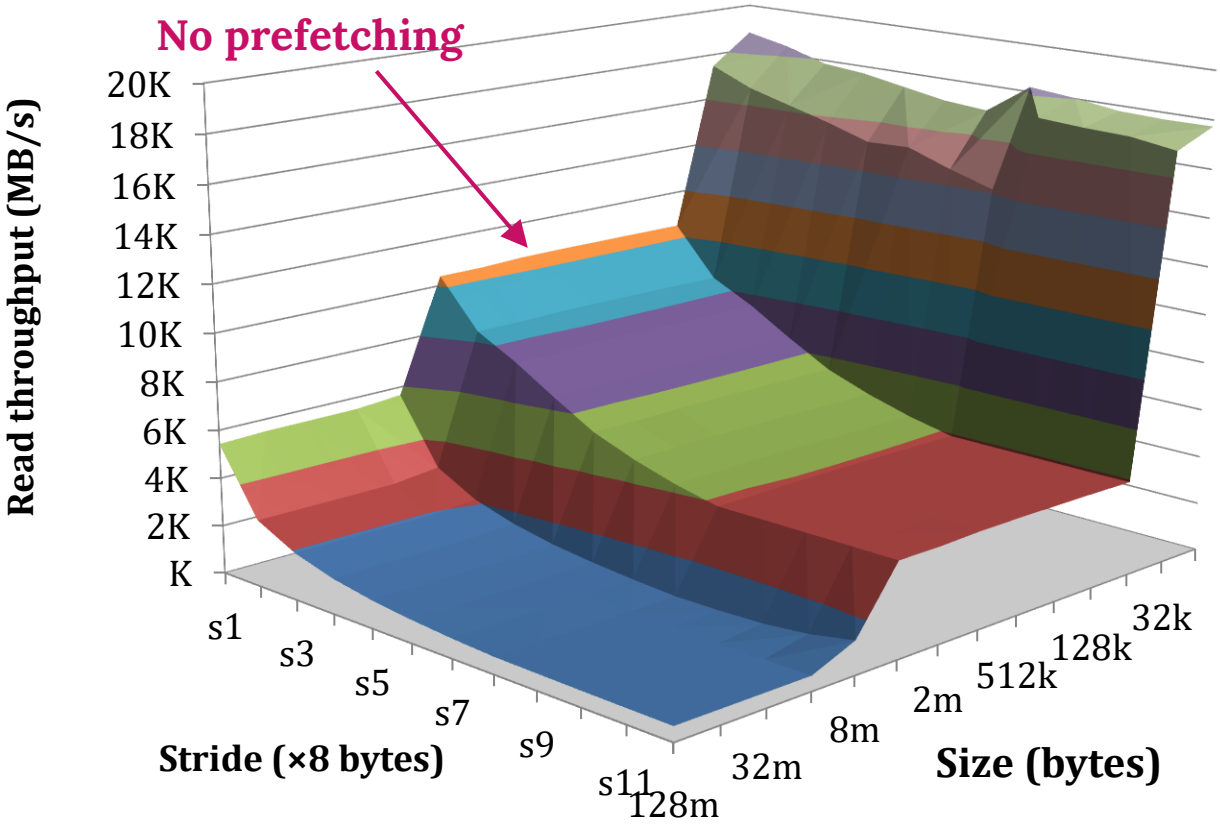
Modeling Block Size Effects from Mem. Mountain

- Core i7 Haswell
  - 2.1-GHz clock freq.
  - 32-KB L1 d-cache
  - 256-KB L2 cache
  - 8-MB L3 cache
  - 64-B block size



2008 Memory Mountain

- Core 2 Duo
  - 2.4-GHz clock freq.
  - 32-KB L1 d-cache
  - 6-MB L2 cache
  - 64-B block size



[CSED211] Introduction to Computer Software Systems

Lecture 11: Cache Memory

Prof. Jisung Park

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

CAOS
COMPUTER ARCHITECTURE & OPERATING SYSTEMS LABORATORY

2023.11.08